Exploring Process Calculi as a Mechanism to Define Dynamic Enumeration Strategies in Constraint Programming

Eric Monfroy[1], **Carlos Olarte**[2] **and Camilo Rueda**[2]
[1] Universidad Técnica Federico Santa María, Valparaíso, Chile
[2] Pontificia Universidad Javeriana, Cali, Colombia
eric.monfroy@univ-nantes.fr, {caolarte,crueda}@atlas.puj.edu.co

Abstract

Constraint programming (CP) has been extensively used to solve a wide variety of problems. Its declarative flavor makes possible to state conditions over variables and the solver finds solutions by applying generic and complete techniques. The process of computing a solution in CP consists mainly in two phases: propagation in which values that are not consistent w.r.t. the constraints are eliminated, and enumeration that chooses a variable and a value for this variable to continue the search when no further propagation is possible. Constraint based languages offer a set of static enumeration strategies. The strategy chosen may affect drastically the time required to find a solution. In this paper we propose a framework to model dynamic enumeration strategies using a stochastic, non-deterministic timed concurrent constraint calculus. Thanks to the reactivity of the calculus, we are able to express strategies that dynamically change according to results observed. Additionally, the compositional approach of the calculus enables us to integrate external knowledge to adapt the strategy. In particular, we integrate knowledge from an incomplete solver to guide the enumeration process. Finally, strategies proposed are integrated with a constraint solver to make "good choices" when it explores the search tree allowing to find solutions quicker.

Keywords: `sntcc` , Constraint programming, Dynamic enumeration strategies

Resumen

La programación por restricciones ha sido utilizada para resolver una gran cantidad de problemas. Su modelo declarativo permite fijar condiciones sobre las variables y un solver genérico encuentra de manera automática una solución. El proceso de calcular una solución en este paradigma consiste en dos fases básicas: Propagación en la que los valores inconsistentes son eliminados del dominio de las variables y enumeración en la que se escoge una variable y un valor para ella y así continuar el proceso cuando no es posible eliminar mas valores del dominio. Los lenguajes basados en este paradigma ofrecen una serie de estrategias estáticas de enumeración. Sin embargo el uso de una estrategia afecta considerablemente el tiempo requerido para encontrar una solución. En este artículo proponemos un framework basado en un cálculo estocástico, no determinístico, concurrente por restricciones para modelar estrategias dinámicas de enumeración. Gracias a que el cálculo es reactivo, las estrategias son capaces de adaptarse de acuerdo a los resultados obtenidos en el proceso de solución. Adicionalmente, gracias al operador de composición paralela, podemos adicionar nueva información del entorno para guiar de mejor manera la búsqueda.

Palabras claves: `sntcc` , Programacion por Restricciones, Estrategias Dinamicas de enumeracion

1 Introduction

The resolution of constraint satisfaction problems (CSP) appears nowadays as a very active and growing research area. Indeed, constraint modeling allows both scientists and practitioners to handle various industrial or academic applications (e.g., scheduling, timetabling, boolean satisfiability, etc.). In this context, CSP are basically represented by a set of decision variables and a set of constraints among these variables. Each variable is associated to a domain which represents the values the variable could be given. The purpose of a resolution process is therefore to assign to each variable a value from its domain such that the constraints are satisfied.

Complete solvers are able to provide the whole set of solutions or to demonstrate that the given problem is unsatisfiable. Complete solvers usually build a search tree by applying domain reduction, splitting and enumeration. Local consistency mechanisms [29] allow the algorithms to prune the search space by deleting inconsistent values from variables domains.

Solver based on constraint propagation is one of the most common methods for solving CSPs. This technique is a complete approach that interleaves splits (e.g., enumeration) and constraint propagations. Constraint propagation prunes the search tree by eliminating values that violate a constraint, i.e., values that cannot participate in any solution of the CSP.

Split generally consists in splitting the domain of a variable into sub-domains. Hence, several smaller CSPs are created from the initial one. Although all strategies of split that preserve the solution set are valid, they have drastically different impacts on resolution efficiency. Moreover, no strategy is (one of) the best for all problems. The issue for efficiency w.r.t. split strategy is thus to be able to select the right value of the right variable.

This problem can be tackled with different approaches. Some studies define some split strategies for generic solvers. In this case, the idea is to develop a generic strategy independent from the problems, and rather efficient on average. Another approach consists in designing a strategy for a class of problems (see e.g., [8]). This time, the strategy relies on the problem structure. The strategies can be static or dynamic. In case of static strategies, the split mechanism is defined before the solving process, based on the structure of the problem. On the other hand, dynamic strategies try to adapt and modify the splitting mechanism based on the behavior of the solving process, on the progress of the resolution, or/and on the evolution of the problem during resolution. Another distinction that can be made is "prediction" or "reparation". Some methods try to predict what is the "best" split strategy to apply next, whereas others try to "repair" or "change" a strategy that was applied previously during the solving process and considered to behave poorly.

We are interested in modeling strategies using a stochastic, non-deterministic concurrent constraint process calculus, the `sntcc` calculus [4], a stochastic extension of `ntcc` [19]. The advantages of using `sntcc` are the following. First, we obtain a clear, formal, and homogeneous (using templates) mobilization of the strategies. Then, the reactive aspect of the calculus 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 knowledge to guide the search when this kind of knowledge is available. The stochastic aspects of the calculus enables us to rank the available split strategies with some probabilities of being applied: the higher the probability, the higher the strategy is judged to be efficient. The non-determinism of the calculus is used to tie-break strategies that have equal probabilities of being applied; we thus introduce randomization from which the solving process can benefit (see, e.g., [14]).
Using `sntcc` , we show several types of split strategies. The first one imposes a sequence of strategies to be used; the sequence of split (interleaved with domain reduction) is repeated until the problem is solved. The aim of this strategy is to diversify the way value selection is performed. The second strategy tries to reward good strategies: after the application of a strategy, its work is evaluated based on some observations of the solving process. If the work is evaluated well, the strategy is rewarded (its probability of being applied is increased), otherwise it is given a penalty (its probability is decreased, and thus it would have less chance to be applied). The third strategy uses an incomplete but very fast solver (i.e., local search) to determine which is the best value to consider. This strategy leads to a hybrid resolution mechanism in which a complete solver is assisted by an incomplete solver to take the split decisions. Finally, a strategy to select the variable to be splited is proposed based on ideas of [14]. This strategy chooses non-deterministically a variable when there are several variables with the same domain size.
This mechanism has been implemented in the Oz language [16] running a simulator of `sntcc` . The experimental results we obtained are more than promising. The dynamic strategies behave better than when considering only one fixed strategy during the whole resolution process.

This paper is organized as follows. Section 2 gives an overview about constraint programming, dynamic enumeration strategies and process calculi. Section 3 describes our framework of dynamic enumeration strategies modeled with `sntcc` calculus. A generic process describing strategies is presented and four instantiations of the framework are proposed. Section 4 is devoted to show some tests using dynamic and static strategies and comparing their performances. Finally, Section 5 concludes the paper and gives some research directions.

2 Backgrounds

2.1 Split strategies

In constraint programming (CP), problems are modeled by means of a set $V = \{v_1, v_2, ..., v_n\}$ of variables, a set of domains for these variables $D = \{D_1, D_2, ..., D_n\}$ and a set of constraints $C = \{c_1, ..c_m\}$. Constraints can be viewed as relations between variables, for example, $c_1(v_1, v_2) = v_1 < v_2 = \{d_1, d_2 | d_1 \in D_1, d_2 \in D_2 \wedge d_1 < d_2\}$. The idea is to choose a value from D for each variable in V s.t all the constrains in the problem are satisfied. Solvers based on constraint propagation alternate phases of pruning of the search tree and phases of split of the search space. The former intends to impose constraints by pruning variable domains, i.e., values that cannot be part of a valid solution are eliminated. In our previous example, if $d_1 = \{2, 3, 4, 6\}$ and $d_2 = \{1, 3, 5\}$, c_1 can be imposed by eliminating 6 from d_1 and 1 from d_2. Propagation is not a complete mechanisms. In some cases, the propagation phase neither can find a solution nor determine if there is no one (e.g. $d_1 = \{2, 3, 4\}$ and $d_2 = \{3, 5\}$ is not a solution for our problem). In this case a split phase is required. This phase creates two or more subproblems to continue the search. For example, we can create the subproblem where $v_1 = 2$ and another one where $v_1 \neq 2$. In each subproblem we can apply again domain reduction. In this way, searching for a solution in CP leads to a search tree where each node represents a new subproblem.
Two main classes of split can be considered: enumeration (the domain is split in one value, and the rest of the domain) and segmentation (e.g., bisection) (the domain is split in several sub-domains). Although all strategies of split preserving solutions are valid, they drastically rule the efficiency of the solving process. However, their effect is very difficult to predict and in most cases, unpredictable. Moreover, no strategy can be considered the (or one of the) best strategy for all kind of problems. The issue is thus to select a variable, and then to decide how to split its domain: for segmentation (how many sub-domains, and where to split) or which value of its domain to enumerate.

It is thus important to select a good strategy, or to avoid a bad one: the minimum value of the domain, the maximum value, the middle value, the value that gives the best outcome of the objective function in case of optimization, etc. Once again, these selections will have totally different effects on the efficiency of the enumeration, e.g., the middle value is well suited for the N-queen problem but it behaves poorly in the magic square case. Depending on the strategy, the difference of time to reach a first solution can be of several orders of magnitude (see, e.g., [9] for some resolution examples using different strategies). 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 defining some generic criteria (e.g., minimum domain) for variable selection, and for value selection (e.g., lower bound, or bisection). The idea besides these works is to obtain a strategy that behaves well on average.

Some other works define specialized strategies tuned to some problems. For some applications, such as Job-Shop Scheduling, specific split strategies have been proposed [8].

Determining the best strategy can be addressed by observing some static criteria. In this case, the value and variable selection criteria is determined once before the resolution process starts (see e.g., [1] for variable ordering selection, [12] to pre-determine the "best" heuristic, [13] to determine the best solver). However, it is well-known that 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 (see e.g., [6] for algorithm control with low-knowledge, [28] for dynamic change of propagators, [10] for variation of strength of propagation).

[2, 3], proposes adaptive constraint satisfaction: algorithms that behaves poorly are detected and dynamically changed by the next candidate in the sequence of algorithms.

In [14, 17] the split strategy is fixed. However, when several choices are ranked equally by the strategy (e.g., there are several "smallest" variables) randomisation is applied for tie-breaking. Moreover, a restart policy based on a specified number of backtracks (cutoff) is also introduced: when the cutoff is reached, the algorithm is restarted at the root of the search tree to try another branch and thus, diversify the solving process.

In [9], instead of predicting what is the best strategy to apply, the authors try to eliminate bad strategies by observing the solving process. The mechanism works as follows: some observations of the solving process are used to draw some indicators; this indicators are then used to evaluate the strategies. Bad strategies are eliminated whereas good one are given more chance to be applied.

2.2 Process Calculi

Process calculi are mathematical formalisms to model and verify concurrent systems. With a few amount of operators, they are able to express a wide variety of behaviors such as mobility [22], time and reactivity [24, 19], process distribution [19], stochastic and probabilistic choices [21, 20, 4, 15], biological function [23, 7] among others. In this paper, we are interested in calculus derived from the Concurrent Constraint model (cc) [25]. cc is based on the concept of constraint as an entity carrying partial information, i.e., conditions for the values that variables can take. This model has been extended to model the notion of time in `tcc` ([24]), non-determinism and asynchrony in `ntcc` ([19]) and stochastic and probabilistic behavior in `pcc` and `sntcc` ([15, 4]). The next sections are devoted to recall the concept of *constraint system* and give a brief description of the `sntcc` calculus.

2.3 Constraint System

In the cc model process interactions can be determined by partial information (i.e. constraints) accumulated in a global store. The particular type of constraints is not fixed but specified in a *constraint system* that is considered a parameter of the calculus.
A constraint system provides a signature from which constraints can be constructed. It also provides an entailment relation ($\models$) over constraints where $c_1 \models c_2$ holds iff the information of c_2 can be inferred from c_1.
Formally, a constraint system is a tuple $\langle \sum, \Delta \rangle$ where $\sum$ is a signature (i.e. a set of constants, functions and predicate symbols) and Δ is a consistent first-order theory over $\sum$ (i.e. a set of sentences over $\sum$ having at least one model). Constraints can be viewed as first-order formulae over $\sum$ and $c \models d$ holds if the implication $c \Rightarrow d$ is valid in Δ [19]. For practical reasons the entailment relation must be decidable.
A constraint *store* is a set of variables and a conjunction of formulae (the 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 of the calculus. For example, $tell(x < 2)$ adds constraint $x < 2$ to the store. Additionally, it is possible to test if a constraint c can be entailed from the store by means of so-called *ask* operations. For example, $ask(x < 5)$ tests whether it is possible to infer that $x < 5$ from the information contained in the *store* (i.e. $store \models x < 5$). The *ask* operation blocks when neither $store \models x < 5$ nor $store \models \neg(x < 5)$ holds.

2.4 `sntcc`

`sntcc` [4] is a process calculus that extends `ntcc` to model stochastic behavior. In both of them, processes share a common store of partial information [26]. In `ntcc` time is conceptually divided into *discrete intervals or time-units*. In a particular time interval, 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 it reaches its resting point (i.e. no further evolution is possible), it responds to the environment with the resulting store. The resting point also determines a residual process which is then executed in the next time interval. `ntcc` has been successfully used to model many real life system such as reactive systems [19], robot behavior [18] among others and `sntcc` has been used mainly modeling biological systems ([5]) where notion of stochastic behavior is fundamental. Unlike `tcc` , `ntcc` includes constructs for modeling *nondeterminism* and *asynchrony*. A very important benefit of being able to specify non-deterministic and asynchronous behavior arises when we are modeling the interaction among several components running in parallel, where each component is part of the environment of the others.

2.5 Process Syntax

In what follows, we show the syntax of `sntcc` operators (including those inherited from `ntcc`). Refers to [4] for a more complete description of the calculus.

- **tell** c: Adds constraint c to the store.

$TELL\dfrac{}{\langle \textbf{tell } c, d\rangle \rightarrow \langle \textbf{skip}, d \wedge c\rangle}$	$SUM\dfrac{}{\langle \sum_{i\in I} \textbf{ when } c_i \textbf{ do } P_i, d\rangle \rightarrow \langle P_j, d\rangle}$
$PAR\dfrac{\langle P, c\rangle \rightarrow \langle P', d\rangle}{\langle P\|\|Q, c\rangle \rightarrow \langle P'\|\|Q, d\rangle}$	$REP\dfrac{}{\langle !P, d\rangle \rightarrow \langle P\|\|\textbf{next } !P, d\rangle}$
$UNLESS\dfrac{}{\langle \textbf{unless } c \textbf{ next } P, d\rangle \rightarrow \langle \textbf{skip}, d\rangle} if \quad d \models c$	$STAR\dfrac{}{\langle \star P, d\rangle \rightarrow \langle \textbf{next }^n P, d\rangle} if \quad n \geq 0$
$RHOP\dfrac{}{\langle {}_\rho P, d\rangle \rightarrow \langle P, d\rangle} if \Phi(\rho) = 1$	$PSUM\dfrac{}{\langle P +_\rho Q, d\rangle \rightarrow \langle P, d\rangle} if \Phi(\rho) = 1$

Table 1: `sntcc` operational semantic.

- $\sum\limits_{i\in I}$ **when** c_i **do** P_i: Chooses non-deterministically a process P_i whose guard c_i is entailed by the store.
- $P||Q$: Represents the parallel composition between P and Q.
- **local** x **in** P: Behaves like P but the information of the variable x is local to P, i.e. P cannot see information about a global variable x and processes which are not part of P cannot see the information generated by P about x.
- **next** P: Executes process P in the next time unit (unit-delay).
- **unless** c **next** P: Executes P iff c cannot be entailed by the constraint store *in the current time unit.*
- $!P$: Executes P in all time units from the current one. It can be viewed as $P||$**next** $P||$**next next** $P||$...
- $\star P$: Represents unbounded but finite delays, i.e. P eventually will be executed. This process can be viewed as $P +$ **next** $P +$ **next next** P...**next** ^nP where n is a finite natural number.
- ${}_\rho P$: P is executed under a probability ρ
- $P +_\rho Q$: Represents the probabilistic choice between P and Q, i.e. P will be chosen with a probability ρ and Q with a probability $1 - \rho$. In this case we use the notation $P +_\rho Q$ as a shorthand for **when** *true* **do** $P +_\rho$ **when** *true* **do** Q, i.e. a choice without guards (blind choice).

2.6 Operational Semantic

Table 2.6 summarizes the operational semantic of `sntcc` . This is based on pairs $\langle P, c\rangle$ where P is a `sntcc` process and c the current store. In each rule, we show the resulting process after execute P with store c and the resulting store. Recall that when stability is reached (i.e., no further evolution in the system is possible), another time units is created with an empty constraint store and the *residual* process is executed. *Residual* processes include **next** P processes and **unless** c **next** P when guard c cannot be entailed from the current constraint store. We use *skip* process to designate the empty process. In rule SUM, I is a finite set of indexes and $i, j \in I$. $\Phi(\rho)$ in rules $RHOP$ and $PSUM$ represents a probabilistic function that returns 1 or 0 according to a binomial distribution and the probability ρ.

3 A framework for integrating dynamic strategies

As we see in the previous section, the computation model of `sntcc` is very suitable to model reactive systems. In our case, we intend to model reactive enumeration strategies based on this calculus. Stimulus will be statistics taken from the solver (e.g., depth, number of variable instantiated, etc) and the resulting store in each time-unit will be the strategy that must be taken. Constructs in the calculus will help us to define which is the "*best*" strategy according to previous choices. In section 3.3 we also show that this framework can be used to implement enumeration strategies guided by external solvers such as local search. The compositional approach of the calculus will enable us to integrate incrementally in a consistent way processes telling new information useful to adapt the strategy.

The framework to integrate this kind of dynamic strategies into the solver works as follow: (1) The constraint solver (search engine) creates the first node in the search tree and executes all the propagators until stability is reached. (2) The search engine invokes the enumeration procedure to select the new variable to be splited and the new value for this variable. (3) The enumeration procedure creates a new `sntcc` time-unit. When doing that, it feeds in the `sntcc` store statistical information of the solving process as stimulus (e.g., **tell** ($Depth = 20 \wedge Vars\_no\_instantiated = 6$)). (4) A `sntcc` simulator that executes programs written in the calculus computes the resulting store according to the stimulus and the processes in the strategy definition. The enumeration procedure remains blocked until the computation in the `sntcc` store finishes. (5) After stability, the variable (var) that must be splited and the value (val) that it must take are both entailed from the `sntcc` store. (6) Finally the enumeration procedure as usual creates two choices over the constraint solver: $var = val(var)$ and $var \neq val(var)$ to guarantee completeness. In this way, the search tree is explored using dynamic choices according to a strategy expressed in the `sntcc` calculus.

The following process definition represents a generic strategy in this framework.

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

Update_Probabilities (**U_P**) changes the probabilities of each strategy according to some rules. This process will define the general behavior of the strategy. 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 a **External_knowledge** (**E_K**) process. Additionally, since `sntcc` is reactive, in some time-units one could add on-line new process as stimulus to fix some parameters in the strategy. It could be useful when expert users observe the solving process and they can give some hints about how to improve the search. Finally, **Choice** makes a probabilistic or non-deterministic choice according to information inferred from both **U_P** and **E_K**.

In forthcoming sections, we instantiate the generic process presented above to define some strategies that will be tested in Section 4.

3.1 Applying multiple enumeration strategies in the same problem

The first dynamic strategy that we propose consists in applying three different static value selection: *min* (selects the lower bound of the domain), *max* (selects the upper bound of the domain) and *mid* (selects the element, which is closest to the middle of the domain) in the same problem in a balanced way. It means, we are going to alternate repetitively the value selection using always the variable with the smallest domain (i.e., variable selection is fixed). This strategy allows us to diversify the value selection performed during the search. The model of this strategy in the calculus is as follows:

$$\begin{aligned}
&\mathbf{Choice} \equiv \mathbf{tell}\ (val = min)\ ||\ \mathbf{next\ tell}\ (val = mid)\ ||\ \mathbf{next^2 tell}\ (val = max) || \mathbf{next^3 Choice}\\
&\mathbf{U\_P} \equiv skip\\
&\mathbf{E\_K} \equiv skip\\
&\mathbf{Strategy_1} \equiv \mathbf{Choice}\ ||\ \mathbf{U\_P}\ ||\ \mathbf{E\_K}
\end{aligned}$$

In **Choice** definition we use $\mathbf{next^n\ P}$ as a shorthand for **next next** ... **next** (P)

3.2 Changing the strategy according to observations

Even though the previous strategy is dynamic (changing of value selection), it applies always the same pattern of choices. The strategy that we present in this section changes the probability of applying a value selection according to the results obtained previously. In this case, the number of variables that has not been instantiated yet is the stimulus (static information) that is introduced by the constraint solver. The idea is to increase the probability of some value selection when it achieves good domain pruning in the next node of the search tree. If the selection fails (i.e. next node fails), its probability is decreased giving more chance to other strategies to be applied.
The model of the strategy is as follows:

$$\begin{aligned}
&\mathbf{Init} = \mathbf{tell}\ (VarNI = Vars\_in\_the\_problem \wedge \rho Min = \tfrac{1}{3} \wedge \rho Mid = \tfrac{1}{3} \wedge \rho Max = \tfrac{1}{3})\\
&\mathbf{U\_P} = (\mathbf{when}\ VarNI < VarNI'\ \mathbf{do}\ increase(val')) + (\mathbf{when}\ VarNI \geq VarNI'\ \mathbf{do}\ decrease(val'))\\
&\mathbf{Choice} = \mathbf{tell}\ (val = min)\ +_{\rho Min}\ (\mathbf{tell}\ (val = mid)\ +_{\rho Mid}\ \mathbf{tell}\ (val = max))\\
&\mathbf{E\_K} \equiv skip\\
&\mathbf{Strategy_2} = \mathbf{Init}\ ||\ !\ \mathbf{next}\ (\mathbf{Choice}\ ||\ \mathbf{U\_P}\ ||\ \mathbf{E\_K})
\end{aligned}$$

In the above expression, $VarNI$ (resp. $VarNI'$) represents the number of variables not instantiated in the current node of the search tree (resp. previous node). $Vars\_in\_the\_problem$ is the number of variables not instantiated after the first propagation step (i.e. when the first node in the tree becomes stable). ρMin, ρMid and ρMax are the probabilities of choosing $val = min$, $val = mid$, and $val = max$, respectively. $increase(val')$ (resp. $decrease(val')$) increases (resp. decreases) the probability ρMin, ρMid and ρMax according to the previous choice val' (val in the previous time unit). Those procedures increase or decrease the probability of the previous selected value in a constant factor and modify the rest of probabilities to guarantee that $\rho Min + \rho Mi + \rho Max = 1$. 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.

3.3 Integrating local search in the enumeration process

Integrating complete and incomplete methods for constraint solving has been shown as a promising field to reduce the time required to obtain solutions [11] in CP. In this case, we use incomplete methods (e.g. local search) to determine which is the best value selection. As first choices (i.e. value selected in the top of the search tree) may affect considerably the number of nodes to be explored during the search, we integrate a process that chooses the value to be splited according to the result computed by local search. Let $V = \{v_1, v_2, ..., v_n\}$ be the set of variables in the problem, with domain $D = \{d_1, d_2, ..., d_n\}$ and $v_i \in V$ s.t $\forall_{v \in V} |dom(v_i)| \leq |dom(v)|$. The processes $LS(vi, value, cv)$ computes local search with a neighborhood function $NB : D \to 2^D$ s.t for all configuration 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. After a fixed number of iterations to avoid time overloads, LS process binds cv to the number of constraint violated. 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 to know which is the most promising assignment to start the search. The following strategy uses this approach to make the first choice and next it continues the search using $Strategy_1$ or $Strategy_2$:

Init = **tell** $VarNI = Vars\_in\_the\_problem$
E_K = $LS(v_i, min(v_i), ls_{min}) || LS(v_i, mid(v_i), ls_{mid}) || LS(v_i, max(v_i), ls_{max})$
Choose = **when** $ls_{min} \leq ls_{mid} \wedge ls_{min} \leq ls_{max}$ **do tell** $(val = min)$
+ **when** $ls_{mid} \leq ls_{min} \wedge ls_{mid} \leq ls_{max}$ **do tell** $(val = mid)$
+ **when** $ls_{max} \leq ls_{min} \wedge ls_{max} \leq ls_{mid}$ **do tell** $(val = max)$
U_P $\equiv$ *skip*
Strategy$_3$ = **Init** || **External – knowledge** || **Choose** || **next** $(Strategy_{1/2})$

This strategy may be easily modified to apply more local search based decisions in the search tree or to update probabilities such as in $Strategy_2$.

3.4 Randomized variable Selection

In previous strategies, the value of the variable is dynamically determined and the variable to be splited is always the variable whose domain is the smallest. In some cases, multiple variables have the same domain size. This strategy aims to randomly (non-deterministically) choose a variable when we have more than one variable with the same domain size (the smallest one). Doing that, we are able to tie-break strategies (variable selections) with the same probability of being applied. This mechanism has shown good performances for some problems (see e.g., [17]).
To implement this strategy, it is necessary to receive as stimulus the state of the variables in the solver, i.e., the domains of the variables. In `sntcc` store we thus have "a copy" of the variables of the solver. Assuming that variables are numbered (e.g., tuples in Oz), this strategy will compute the minimal size of the variables (min_size) domain to filter the set of variables whose size is equal to min_size. This new list of variables will be stored in a local variable $VSet$. Next, the strategy choose non-deterministically one variable in the set $VSet$. The complete model is presented as follows:

U_P = **tell** $(Min = min\_domain(Problem\_vars))$||**tell** $(VSet = filter(Problem\_vars, Min))$
Choose = $\sum_{v_i \in VSet}$ **when** *true* **do tell** $(Var = v_i)$
Strategy$_4$ =! **local** $Min, VSet$ **in Choose** || **Update**

	$Strategy_1$		$Strategy_2$		$Strategy_3$		Max		Min		Mid	
	Nodes	Time(s)	Nodes	Time	Nodes	Time	Nodes	Time	Nodes	Time	Nodes	Time
N=3	7	0.4	6	0.4	7	0.4	16	0.1	27	0.09	6	0.09
N=4	1729	17.7	7	0.42	19	1.0	1209	0.13	15	0.11	380	0.16
N=5	764	7.9	91	11.4	318	4.7	-	-	158006	10.4	151211	11.11
N=6	136	1.6	1327	7.61	278	6.2	-	-	-	116m.46s	-	-
N=7	1024	10.7	142	2.1	459	10.9	-	-	-	-	-	-
N=8	149	1.9	65	7.2	3116	41.1	-	-	-	-	-	-

Table 2: Number of nodes explored. Magic Square Problem

	$naive$		$size$		min		max		$Strategy_4$	
	Nodes	Time(s)	Nodes	Time	Nodes	Time	Nodes	Time	Nodes	Time
N=8	8	0.3	8	0.3	23	0.31	9	0.3	4	0.35
N=16	22	0.3	70	0.31	94	0.31	40	0.3	11	0.45
N=32	119636	4.1	79	0.32	-	-	-	-	26	0.57
N=64	-	-	89	0.4	-	-	-	-	58	0.8
N=128	-	-	127	0.42	-	-	-	-	124	1.61
N=256	-	-	257	0.39	-	-	-	-	255	3.44

Table 3: Number of nodes explored. N-queens Problem

Notice that this strategy only performs variable selection. We can execute it in parallel with the previous strategies to integrate both choices (i.e. variable and value selection) during the solving process.

4 Results

All 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. $Strategy_1$, $Strategy_2$ and $Strategy_3$ (using $Strtegy_1$ after the local search-based decision) were tested with the canonical problem *Magic Square*. This problem consists in finding an $N \times N$-matrix such that every field of the matrix is an integer between 1 and N^2, the fields of the matrix are pairwise distinct and the sums of the rows, columns, and the two main diagonals are all equal. In each instance, we measured the number of nodes created using our strategies and the fixed strategies Max, Min and Mid provided by Oz. All of them always select the variable with the smallest domain. Table 2 shows the results. In this table "-" denotes more than one million of nodes. Each row in the table represents an instance of the problem. For dynamic and static strategies we show the number of nodes of the search tree (including failed nodes) and the time (in seconds) necessary to find a solution. Notice that dynamic enumeration strategies outperform static enumerations in each case according to the number of nodes in the search tree. Nevertheless, for small instances (i,e $N \leq 4$) the time spent by dynamic strategies is higher than time taken for static strategies because of the overload executing `sntcc` process.
To test $Strategy_4$ we use the *N-Queen* Problem. This problem consists in placing N queens on an $N \times N$ chess board such that no two queens attack each other (see [27]). Table 3 summarize the results. We compare this strategy with fixed strategies: *naive* (selects the leftmost variable), *size* (selects the leftmost variable whose domain is minimal), *min* (selects the leftmost variable whose lower bound is minimal.) and *max* (selects the leftmost variable whose upper bound is maximal) provided by Oz. In all the cases the value selected is the closest to the middle of the domain (*mid*). Notice that *size* strategy is a good strategy exploring a few amount of nodes before finding a solution. Nevertheless $Strategy_4$ makes some improvements randomizing the variable selection when multiple variables have the same domain size. Notice also that since differences between $Strategy_4$ and *size* w.r.t the number of nodes are small, *size* strategy is better in terms of running time because of the time required to compute time-units in $Strategy_4$.

5 Concluding remarks

A generic framework to model and integrate dynamic enumeration strategies in a constraint propagation-based solver was proposed in this paper. This framework is based on a stochastic non-deterministic concurrent constraint calculus, the `sntcc` calculus. The reactivity of the calculus allows us to adapt the strategy according to observations (indicators) taken from the process resolution. Additionally, parallel composition operator in the calculus enables us to add new process to guide the search. These new process can be external process such as incomplete methods (e.g. local search) or heuristics from expert-users. The framework was instantiated in four dynamic strategies. The first one diversified the value selection. The second one used a set of probabilities to reward good strategies and penalize bad ones. The third one included local search process to obtain the most promising branch in the search tree. Finally, the last strategy aim to randomize the variable selection when more than one variable had the shortest domain. All of this strategies showed a better performance than using a single static strategy to solve two well known constraint satisfaction problems. In the case of Magic Square problem, $Strategy_1$, $Strategy_2$ and $Strategy_3$ outperformed static value selection *min*,*mid* and *max*. By the other side, $Strtegy_4$ made some improvements to static variable selection *size* (select the left most variable with smallest domain) in the N-Queen problem.

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 (evolution) provided by the calculus. By the other side, 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. For example, $Strategy_2$ can behave poorly when it changes repetitively the probabilities around the same values, i.e $\rho_{min_i} \approx \rho_{min_{i+1}} \approx ... \approx \rho_{min_{i+n}}$ where ρ_{min_i} represents ρ_{min} in the i^{th} time-unit. This misbehavior can be detected by a `sntcc` process running in parallel with $Strategy_2$ and solved by changing radically the value of ρ_{min} (e.g increasing it near to 1 or decreasing it near to 0) in the next time-unit. This move allows to "escape" from this kind of configurations. 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. First, try to integrate more indicators from the solving process to guide in a better way the search. By the other side, to integrate more external solver and achieve coordination by using process calculi. The final idea is to provide a generic framework for hybrid resolution of CSPs.

References

[1] J. C. Beck, P. Prosser, and R. Wallace. Variable Ordering Heuristics Show Promise. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming, CP'2004*, volume 3258 of *Lecture Notes in Computer Science*, pages 711–715, 2004.

[2] J. Borrett, E. Tsang, and N. Walsh. Adaptive constraint satisfaction. In *15th UK Planning and Scheduling Special Interest Group Workshop*, Liverpool, 1996.

[3] James E. Borrett, Edward P. K. Tsang, and N. R. Walsh. Adaptive constraint satisfaction: The quickest first principle. In *Proceedings of 12th European Conference on Artificial Intelligence, ECAI'1996*, pages 160–164. John Wiley and Sons, 1996.

[4] Olarte C. and Rueda C. A stochastic non-deterministic temporal concurrent constraint calculus. In *XXV International Conference of the Chilean Computer Science Society, SCCC 2005*, 2005.

[5] Olarte C. and Rueda C. Using stochastic ntcc to model biological systems. In *XXXI Conferencia Latinoamericana de Inform'atica, CLEI 2005*, 2005.

[6] 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.

[7] L. Cardelli. Brane calculi. In *CMSB*, pages 257–278, 2004.

[8] Y. Caseau and F. Laburthe. Improved clp scheduling with task intervals. In *Proceedings of the International Conference on Logic Programming, ICLP'1994*, pages 369–383. MIT Press, 1994.

[9] Carlos Castro, Eric Monfroy, Christian Figueroa, and Rafael Meneses. An approach for dynamic split strategies in constraint solving. In *Proceedings of MICAI'05 : Advances in Artificial Intelligence, 4th Mexican International Conference on Artificial Intelligence, Monterrey, Mexico*, volume 3789 of *Lecture Notes in Computer Science*, pages 162–174. Springer, 2005.

[10] H. El Sakkout, Mark Wallace, and Barry Richards. An instance of adaptive constraint propagation. In *Proceedings of the Int. Conference on Principles and Practice of Constraint Programming, CP'96*, volume 1118 of *Lecture Notes in Computer Science*, pages 164–178. Springer, 1996.

[11] Frdric Saubion Eric Monfroy and Tony Lambert. Hybrid csp solving. In *Focos 2005 (Invited paper)*, 2005.

[12] Pierre Flener, Brahim Hnich, and Zeynep Kiziltan. A meta-heuristic for subset problems. In *Proceedings of Practical Aspects of Declarative Languages, PADL'2001*, volume 1990 of *Lecture Notes in Computer Science*, pages 274–287. Springer, 2001.

[13] Cormac Gebruers, Alessio Guerri, Brahim Hnich, and Michela Milano. Making choices using structure at the instance level within a case based reasoning framework. In *Proceedings of CPAIOR'2004*, volume 3011 of *Lecture Notes in Computer Science*, pages 380–386. Springer, 2004.

[14] Carla Gomes, Bart Selman, and Henry Kautz. Boosting combinatorial search through randomization. In *Proceedings of AAAI'98*, pages 431–437, Madison, Wisconsin, 1998.

[15] Vineet Gupta, Radha Jagadeesan, and Vijay A. Saraswat. Probabilistic concurrent constraint programming. In *International Conference on Concurrency Theory*, pages 243–257, 1997.

[16] Seif Haridi and Nils Franzn. *Tutorial of Oz.*, 2004. Available at `www.mozart-oz.org`.

[17] H. Kautz, E. Horvitz, Y. Ruan, C. Gomes, and B. Selman. Boosting combinatorial search through randomization. In *Proceedings of AAAI'2002*, pages 674–682, 2002.

[18] Pilar Munoz and Andres Hurtado. Programming robot devices with a timed concurrent constraint programming. In *Principles and Practice of Constraint Programming - CP2004. LNCS 3258*, page 803. Springer, 2004.

[19] Mogens Nielsen, Catuscia Palamidessi, and Frank D. Valencia. Temporal concurrent constraint programming: Denotation, logic and applications. In *Special Issue of Selected Papers from EXPRESS'01, Nordic Journal of Computing*, 2001.

[20] Alessandra Di Pierro and Herbert Wiklicky. An operational semantics for probabilistic concurrent constraint programming. In *International Conference on Computer Languages*, pages 174–183, 1998.

[21] C. Priami. Stochastic pi-calculus. In *Computer Journal*, 2004.

[22] J. Parrow R. Milner and D. Walker. A calculus of mobile processes, Parts I and II. *Journal of Information and Computation*, 100:1–77, September 1992.

[23] A. Regev, E. M. Panina, W. Silverman, L. Cardelli, and E. Y. Shapiro. Bioambients: an abstraction for biological compartments. *Theor. Comput. Sci.*, 325(1):141–167, 2004.

[24] V. Saraswat, R. Jagadeesan, and V. Gupta. Fundation of timed concurrent constraint programming. In *IEEE Symposium on Logic in Computer Science.* IEEE press, 1994.

[25] V. Saraswat, M. Rinard, and P. Panangaden. Semantic foundations of concurrent constraint programming. In *18th Annual ACM Symposium on Principles of Programming Languages*, pages 333–353. ACM Press, 1991.

[26] V. A. Saraswat. *Concurrent Constraint Programming.* The MIT Press, Cambridge, MA, 1993.

[27] C. Schulte and G. Smolka. *Finite Domain Constraint Programming in Oz. A Tutorial.*, 2004. Available at `www.mozart-oz.org`.

[28] C. Schulte and P. J. Stuckey. Speeding up constraint propagation. In *Proceedings of International Conference on Principles and Practice of Constraint Programming, CP'2004*, volume 3258 of *Lecture Notes in Computer Science*, pages 619–633, 2004.

[29] E. Tsang. *Foundations of Constraint Satisfaction.* Academic Press, London, 1993.