A Fair Extension of (Soft) Concurrent Constraint Languages

Stefano Bistarelli[1,2,3] and Paola Campli[1]

[1] Department of Science, University "G. d'Annunzio" of Chieti-Pescara, Italy
[bista,campli]@sci.unich.it
[2] Departement of Maths and Informatics, University of Perugia, Italy
bista@dipmat.unipg.it
[3] Institute of Informatics and Telematics (IIT), CNR Pisa, Italy
stefano.bistarelli@iit.cnr.it

Abstract. The aim of this paper is to guarantee fair computations in (Soft) Concurrent Constraint languages. We present an extension of the semantics related to the operator of parallelism in order to allow a "fair" selection for the execution of concurrent agents. We define a new operator of Parallelism ($\|_m$) which is able to deal with a finite number (m) of agents. Subsequently we apply the general rule to different fairness notions.

Introduction

This paper contains an extension of the semantics of both Concurrent Constraint (CC) and Soft Concurrent Constraint (SCC) languages related to the operator of parallelism; the aim is to guarantee a fair criterion of selection among parallel agents and to restrict or remove the possible unwanted behavior of a program. In fact, the classical (S)CC parallel operator selects one agent among more parallel enabled agents at time. As the parallelism rule does not provide any criterion about the choose, some agents could be executed more time with respect to the others, or could not be executed at all. We provide a general rule to express fairness with a finite number of agents in the CC language; moreover we apply the rule in three particular notions of fairness: *Carpooling-fairness*, *Weak h-fairness* and *Strong h-fairness* (where the notion of h-fairness is given by [3]).

1 Background and overview of the existing literature

1.1 Fairness in programming languages

Some of the most common notions of fairness in computer science are given by Nissim Francez in [5]: **weak fairness** requires that if an action (or agent) is *continuously enabled*, so it can almost always proceed, then it must *eventually* do so, while **strong fairness** requires that if an action (or agent) can proceed

infinitely often then it must *eventually* proceed (be executed). These notions are available only for infinite computations.
The notions of fairness for infinite computations can be expressed also for finite computations through the notion of *Bounded Fairness* [3].

1.2 Concurrent Constraint Programming

The concurrent constraint (cc) programming paradigm [7] concerns the behavior of a set of concurrent agents with a shared store, which is a conjunction of constraints. Each computation step possibly adds new constraints to the store. The concurrent agents communicate only with the shared store, by either checking if it entails a given constraint (*ask* operation) or adding a new constraint to it (*tell* operation).

Here is the transition rule of the parallelism (see [6] for the complete CC transition system)

$$\frac{\langle A_1,\sigma\rangle \rightarrow \langle A_1^{'},\sigma^{'}\rangle}{\langle A_1 \| A_2,\sigma\rangle \rightarrow \langle A_1^{'} \| A_2,\sigma^{'}\rangle} \quad \frac{\langle A_1,\sigma\rangle \rightarrow \langle A_1^{'},\sigma^{'}\rangle}{\langle A_2 \| A_1,\sigma\rangle \rightarrow \langle A_2 \| A_1^{'},\sigma^{'}\rangle} \text{ parallelism (1)}$$

The parallelism operator works as follow: if agent A_1 with a store σ evolves in agent A_1' with store σ', then the parallel execution of the agents A_1 and A_2 evolves in the parallel execution between agents A_1' and A_2.
The rule above, does not provide a criterion to establish which agent the scheduler has to select in a parallel execution when only one agent at time can be executed. In fact, it could choose always agent A_1 and never select agent A_2. We want to avoid these possible situations by providing a fair criterion for the selection of concurrent agents. To obtain this, we use a new parallel operator (see Section 2).

Soft Concurrent Constraint. The Soft Concurrent Constraint (SCC) language [2] uses soft constraints that generalize classical constraints by allowing several levels of satisfaction. The framework is based on semirings [1]; the formalism provides suitable operations to combine constraints ($\times$) and to compare ($+$) the tuples of values and constraints.

The $\sqsubseteq$ operator. A partially ordered set is a pair $(D, \sqsubseteq)$ where D is a set and $\sqsubseteq$ is a binary relation over D (i.e. a subset of DxD), such that: $\sqsubseteq$ is reflexive (i.e. $d \sqsubseteq d \quad \forall d \in D$), antisymmetric (i.e. $d \sqsubseteq e$ and $e \sqsubseteq d$ imply $d = e \quad \forall d, e \in D$) and transitive (i.e. $d \sqsubseteq e \sqsubseteq d'$ implies $d \sqsubseteq d' \quad \forall d, d', e \in D$)

2 The new parallel operator

In order to obtain a Fair version of the parallel execution of a finite number of agents, we modify the parallel operator $\|$ in the CC semantic by replacing it

with the $\|_m$ operator, which is able to deal with a set of m agents. Since with the classical parallel operator only two concurrent agents are represented in the Parallelism rule, it is not possible to express fairness among more than 2 agents at the same level.
We develop the new transition rule, which is able to operate with different fairness definitions or scheduling algorithm with finite computations. This is possible by introducing an array $\boldsymbol{k}$ and a set of conditions over $\boldsymbol{k}$, $< cond(k) >$ for the selection of the agent that we want to execute. The array $\boldsymbol{k}$ (with m elements) allows us to keep track of the actions performed by each agent during the computational steps. In this way we can associate a value $\boldsymbol{k}[i]$ to each agent A_i. The rule considers among the m agents, the n enabled agents (that we indicate with the notation: $A_{l_1} \rightarrow, \ldots, A_{l_n} \rightarrow$). This rule can be applied to different notions and metrics of fairness, as it is stated in a general way. According the algorithm or the fairness notion that we want to consider, we can use different assignments for the array $\boldsymbol{k}$. The modified parallelism rule is shown in Table 1.

Table 1. $\|_m$ - parallelism (1) rule

$$\frac{\begin{array}{c} A_{l_1} \rightarrow, \ldots, A_{l_n} \rightarrow \\ < cond(k) > \qquad < A_{li}, \sigma > \rightarrow < A'_{l_i}, \sigma' > \end{array}}{< ||_m(A_1, \ldots, A_{l_i}, \ldots, A_m), \boldsymbol{k}, \sigma > \rightarrow < ||_m(A_1, \ldots, A'_{l_i}, \ldots, A_m), \boldsymbol{k}', \sigma' >}$$

$$\boldsymbol{k}[x]' = \begin{cases} \text{assignment 1} & \text{if } x = 1, \ldots, m \quad \text{for not enabled agents} \\ \text{assignment 2} & \text{if } x = l_i \qquad\quad \text{for the enabled and executed agent} \\ \text{assignment 3} & \text{if } x = l_j \quad j \neq i \text{ for the enabled and not executed agents} \end{cases}$$

In the next sections, we show how to apply this general rule to different fair algorithms or levels of fairness. The first idea we use, in order to provide a way to establish which of the agents can evolve, cames from a *Fair carpooling scheduling algorithm* [4]; subsequently we apply the new rule to obtain Strong h-fairness and Weak h-fairness (Section 2.2).

2.1 Carpooling-fairness

The fair carpool scheduling algorithm. Carpooling consists in sharing a car among a driver and one or more passengers to divide the costs of the trip. We want a scheduling algorithm that will be perceived as fair by all the members as to encourage their continued participation.
Let U a value that represents the total cost of the trip. It is convenient to take U to be the least common multiple of $[1, 2, \ldots, m]$ where m is the largest number of people who ever ride together at a time in the carpool. Since in a determined day the participants can be less than m, we define n as the number of participants in the carpooling in a given day ($n \in [1 \ldots m]$). Each day we calculate the

passengers and driver's scores. In the first day the score is zero for each member; in the following days the driver will increase his score of $U(n-1)/n$, while the remaining $n-1$ passengers decrease their score of U/n.

With reference to this algorithm, we represent the driver with the agent A_{l_i}, while the passengers are the remaining $(n-1)$ enabled agents.
Let $n \leq m$ represents the number of enabled agents and A_{l_i} (with $i = [1, \ldots, n]$) the agent enabled and executed. A_{l_i} increases his score of $\alpha_n = U(n-1)/n$, while the $n-1$ enabled and not executed agents decrease their score of $\beta_n = U/n$. Initially all the m elements of the array $\boldsymbol{k}$ are equal to 0. Subsequently, we add α_n to the previous value ($\boldsymbol{k}[l_i]$) of the agent A_{l_i} and we subtract β_n to the previous value ($\boldsymbol{k}[l_j]$) of the other agents $A_{lj} \quad \forall j \in [1, \ldots, n], \quad j \neq i$.

In this case we use the condition $< cond(k) >= \boldsymbol{k}[l_i] \leq \boldsymbol{k}[l_j]$ that allows the (enabled) agent with a lower score to evolve. We update the scores in the variable $\boldsymbol{k}$ in the following way: the agent A_{l_i} (that is the agent which performs the execution) increases his previous score of α_n; the $(n-1)$ enabled agents decrease their score of β_n, while the scores of the not enabled agents remain unchanged. With the parameters provided by the carpooling algorithm we obtain the Carpooling-fairness rule:

Table 2. Carpooling fairness - parallelism (1) rule

$$\frac{\begin{array}{c} A_{l_1} \rightarrow, \ldots, A_{l_n} \rightarrow \\ \boldsymbol{k}[l_i] \leq \boldsymbol{k}[l_j] \quad \forall \quad j = 1, \ldots, n \quad < A_{li}, \sigma > \rightarrow < A'_{l_i}, \sigma' > \end{array}}{< ||_m(A_1, \ldots, A_{l_i}, \ldots, A_m), \boldsymbol{k}, \sigma > \rightarrow < ||_m(A_1, \ldots, A'_{l_i}, \ldots, A_m), \boldsymbol{k}', \sigma' >}$$

$$\boldsymbol{k}[x]' = \begin{cases} \boldsymbol{k}[x] & \text{if } x = 1, \ldots, m \\ \boldsymbol{k}[x] + \alpha_n & \text{if } x = l_i \\ \boldsymbol{k}[x] - \beta_n & \text{if } x = l_j \qquad j \neq i \end{cases}$$

At each computation we select an agent according its value in the array $\boldsymbol{k}$ and then we update it in a way that at the following step, another agent (possibly) has to be executed.

2.2 Bounded-fairness

In order to make the notion of (strong and weak) fairness sensible also for finite computations, we parametrise the definition with a value h and we ask that *an event which is (infinitely often or continuously) enabled at least h times, must necessarily be executed* (this definition is based on [3]). Therefore we combine the notion of h-fairness with the classical definition of Weak and Strong fairness [5] to obtain *Weak h-fairness* and *Strong h-fairness*. These two different levels of fairness are also available for finite computations.

Strong h-fairness. Strong h-fairness follows from strong and h-fairness definitions and it requires that *if an agent (or event) is enabled at least h times, it will be eventually executed*. To guarantee this statement in the parallelism rule we can say that whatever the value h is, we want to execute the event which is enabled many times with respect to the others.
To obtain this, the array $\boldsymbol{k}$ will contain the times an agent is enabled; we set to 0 the value $\boldsymbol{k}[l_i]$ of the agent A_{l_i} which is executed and we increase of 1 the values $\boldsymbol{k}[l_j]$ (for $j = 1, \ldots, n \quad$ and $j \neq i$) of the enabled agents that are not executed. The values of the not enabled agents $(A_1, \ldots, A_m)$ remains, obviously, unchanged. In this case, to be executed, an enabled agent (A_{l_i}) has to fulfill the condition $< cond(k) >= \boldsymbol{k}[l_i] \geq \boldsymbol{k}[l_j]$, that is, it must be enabled many times than that of the others enabled j agents. The new parallelism rule is represented in table 3.

Table 3. Strong h-fairness - parallelism rule

$$\frac{\begin{array}{c} A_{l_1} \rightarrow, \ldots, A_{l_n} \rightarrow \\ \boldsymbol{k}[l_i] \geq \boldsymbol{k}[l_j] \quad \forall \quad j = 1, \ldots, n \qquad < A_{l_i}, \sigma > \rightarrow < A'_{l_i}, \sigma' > \end{array}}{< ||_m(A_1, \ldots, A_{l_i}, \ldots, A_m), \boldsymbol{k}, \sigma > \rightarrow < ||_m(A_1, \ldots, A'_{l_i}, \ldots, A_m), \boldsymbol{k}', \sigma' >}$$

$$\boldsymbol{k}[x]' = \begin{cases} \boldsymbol{k}[x] & \text{if } x = 1, \ldots, m \\ 0 & \text{if } x = l_i \\ \boldsymbol{k}[x] + 1 & \text{if } x = l_j \end{cases}$$

Weak h-fairness. Weak h-fairness requires that *if an agent is enabled at least h consecutive times, than it must be executed*. This means that whatever the value of h is, we want to execute the agent enabled more consecutive times than the others. In order to guarantee these requirements we extend the general $||_m$ rule by including an array $\boldsymbol{b}$ which contains 0 and 1 values. The i-th element $\boldsymbol{b}[l_i]$ has value 1 only if the agent A_{l_i} was previously enabled. Moreover we use the array $\boldsymbol{k}$ that contains the number of times an agent is consecutively enabled: we increase of 1 the value $\boldsymbol{k}[l_j]$ of an agent A_{l_j} only if it was previously enabled (that is $\boldsymbol{b}[l_j] = 1$) and we set to 0 the value of the agent which performs an execution. More in detail, we select among the enabled agents $(A_{l_1} \rightarrow, \ldots, A_{l_n} \rightarrow)$ the one which fulfills the condition $\boldsymbol{k}[l_i] \geq \boldsymbol{k}[l_j]$, that is, the agent which is enabled consecutively more times than the others. We set to 0 the values of the arrays $\boldsymbol{b}$ and $\boldsymbol{k}$ for all the not enabled agents in the current step; we set to 0 the value $\boldsymbol{b}[l_i]$ of the enabled and executed agent A_{l_i}, and to 1 the values $\boldsymbol{b}[l_j]$ of the remaining enabled agents. If the agents were previously enabled (therefore $\boldsymbol{b}[l_j] = 1$), we increment the values $\boldsymbol{k}[l_j]$, otherwise we set to 1 the value of the currently but not previously (with $\boldsymbol{b}[l_j] = 0$) enabled agent. We obtain the rule in Table 4.

Table 4. Weak h-fairness - parallelism rule

$$\frac{\begin{array}{c} A_{l_1} \rightarrow, \ldots, A_{l_n} \rightarrow \\ \boldsymbol{k}[l_i] \geq \boldsymbol{k}[l_j] \quad \forall \quad j = 1, \ldots, n \qquad < A_{l_i}, \sigma > \rightarrow < A'_{l_i}, \sigma' > \end{array}}{< ||_m(A_1, .., A_{l_i}, .., A_m), \boldsymbol{b}, \boldsymbol{k}, \sigma > \rightarrow < ||_m(A_1, .., A'_{l_i}, .., A_m), \boldsymbol{b}', \boldsymbol{k}', \sigma' >}$$

$$\boldsymbol{b}[x]' = \begin{cases} 0 & \text{if } x = 1, \ldots, m \\ 1 & \text{if } x = l_j \quad \forall j = 1, \ldots, n \end{cases}$$

$$\boldsymbol{k}[x]' = \begin{cases} \boldsymbol{k}[x] & \text{if } x = 1, \ldots, m \qquad x \neq l_j \\ 0 & \text{if } x = l_i \\ \boldsymbol{k}[x] + 1 & \text{if } x = l_j \quad \text{and} \quad \boldsymbol{b}[l_j] = 1 \quad \forall j = 1, \ldots, n \quad j \neq i \\ 1 & \text{if } x = l_j \quad \text{and} \quad \boldsymbol{b}[l_j] = 0 \quad \forall j = 1, \ldots, n \quad j \neq i \end{cases}$$

2.3 Using $\|_m$ for Soft Concurrent Constraint

We can use the new $\|_m$ operator also to deal with soft constraints. In this case, we want to select the agent with a lower level of preference; in the soft case the array $\boldsymbol{k}$ previously defined in the general rule contains soft constraints (unlike values used in the crisp case) added by the Agents which perform a tell action. This array is divided into sections: for each agent A_i there is a section $\boldsymbol{k}[i]$ that contains the agent's constraints. These are combined through the $\otimes$ operator. For example, if Agent i performs the action: $tell(c)$, we will modify the section $\boldsymbol{k}[i]$, obtaining $\boldsymbol{k}[i]' = \boldsymbol{k}[i] \otimes c$. Below we show the modified rule that uses the array of soft constraints, and as condition $< cond(k) >= \boldsymbol{k}[l_i] \sqsubseteq \boldsymbol{k}[l_j]$.

Table 5. $\|_m$ - parallelism (1) with for soft constraints

$$\frac{\begin{array}{c} A_{l_1} \rightarrow, \ldots, A_{l_n} \rightarrow \\ \boldsymbol{k}[i] \sqsubseteq \boldsymbol{k}[j] \qquad < A_{li}, \sigma > \rightarrow < A'_{l_i}, \sigma' > \end{array}}{< ||_m(A_1, \ldots, A_{l_i}, \ldots, A_m), \boldsymbol{k}, \sigma > \rightarrow < ||_m(A_1, \ldots, A'_{l_i}, \ldots, A_m), \boldsymbol{k}', \sigma' >}$$

$$\boldsymbol{k}[x]' \begin{cases} \boldsymbol{k}[x] & \text{if } x = 1, \ldots, m \\ \boldsymbol{k}[x] \otimes c & \text{if } x = l_i \\ \boldsymbol{k}[x] & \text{if } x = l_j \quad j \neq i \end{cases}$$

This rule means that when the agent with the lower level of preference (A_{l_i}) is selected (since the guard $\boldsymbol{k}[l_i] \sqsubseteq \boldsymbol{k}[l_j]$ permits it), it combines its constraint (c) with the element $\boldsymbol{k}[l_i]$ obtaining $\boldsymbol{k}[l_i] \otimes c$ (therefore its level of preference increases) and this constraint is inserted in the store σ; in this way we obtain a new store $\sigma^{'}$ that corresponds to the old store combined with the new constraint: $\sigma^{'} = \sigma \otimes c$.

In the next step another agent is selected and its constraint combined with the previous constraints of the array $\boldsymbol{k}$, then it is added to the store and so on for the successive steps. We can consider this rule as the strong fairness version with soft constraints. Furthermore we can add the array of boolean values to obtain a soft version of the weak fairness rule.

Acknowledgement

We wish to thank Paolo Baldan for his help in discussing and improving the contents of the paper.

3 Future work

We are going to use the modified parallelism rule with metrics based on other fair scheduling algorithms. Subsequently we plan to study the utility functions used in economics as level of fairness, and to associate it to a semiring structure with the aim to measure "*how much the computation is fair*". To do this we plan to use the same indexes that are used to measure the inequality in economics, such as the Gini coefficient. We also plan to use social welfare functions to aggregate individual preferences, in a way that each agent is equally satisfied with respect to the others and according to his preferences.

References

1. S. Bistarelli. *Semirings for Soft Constraint Solving and Programming*, volume 2962 of *Lecture Notes in Computer Science*. Springer, London, UK, 2004.
2. Stefano Bistarelli, Ugo Montanari, and Francesca Rossi. Soft concurrent constraint programming. *ACM Trans. Comput. Logic*, 7(3):563–589, 2006.
3. Nachum Dershowitz, D. N. Jayasimha, and Seungjoon Park. Bounded fairness. In Nachum Dershowitz, editor, *Verification: Theory and Practice*, volume 2772 of *Lecture Notes in Computer Science*, pages 304–317. Springer, 2003.
4. Ronald Fagin and John H. Williams. A fair carpool scheduling algorithm. *IBM Journal of Research and Development*, 27(2):133–139, 1983.
5. Nissim Francez. *Fairness*. Springer-Verlag New York, Inc., New York, NY, USA, 1986.
6. Vijay A. Saraswat. *Concurrent constraint programming*. MIT Press, Cambridge, MA, USA, 1993.
7. Vijay A. Saraswat and Martin Rinard. Concurrent constraint programming. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 232–245, New York, NY, USA, 1990. ACM.