UNIVERSITE D'ORLEANS
Faculté des Sciences
LIFO

Laboratoire d'Informatique Fondamentale d'Orléans
4, rue Léonard de Vinci, BP 6759
F-45067 Orléans Cedex 2
FRANCE

Rapport de Recherche

www : http://www.univ-orleans.fr/SCIENCES/LIFO/

Located Concurrent Constraint Programming

Nicolas ROMERO
Université d'Orléans, LIFO

Rapport N^o **2000-08**

Located Concurrent Constraint Programming

Nicolas Romero

Nicolas.Romero@lifo.univ-orleans.fr

Laboratoire d'Informatique Fondamentale d'Orléans
Université d'Orléans – Bât. IIIA
1, rue Léonard de Vinci
BP 6759
45067 Orléans Cedex 2 – France

Abstract The Located Concurrent Constraint Programming model enhances the Saraswat's Concurrent Constraint Programming framework (`cc`) with distribution- and time-aware primitives. The unique global constraint store is split into several *locations*, which are local stores together with some associated processes. Each one can be dynamically created and moves on its own initiative to be interfaced with others. Multiform time awareness appears with the ability to detect local store modifications, to start and to suspend locations. A formal syntax and its operational semantics are defined. We then discuss several examples to show that these minimal changes lead to a powerful distributed concurrent constraint framework.

Résumé La Programmation Concurrente Située par Contraintes étend le cadre de la Programmation Concurrente par Contraintes développé par Saraswat avec des primitives prenant en compte la distribution et le temps. Le store global unique est éclaté en plusieurs *sites*, qui sont constitués d'un store local et des processus associés. Chacun peut être créé dynamiquement et se déplace de son propre chef pour s'interfacer avec d'autres. La prise en compte du temps multiforme apparaît avec la capacité de détecter des modifications du store local, de démarrer et de suspendre des sites. Une syntaxe formelle et la sémantique opérationnelle associée sont définies. Nous présentons ensuite plusieurs exemple pour montrer que ces ajouts créent un cadre de programmation distribuée par contraites puissant.

1 Introduction

Constraint programming smoothly unifies in a same framework the declarativity inherited from logic programming with the efficiency of computation over specific domains using specialized solvers. Its development has grown considerably during the last decade, and theoretical as well as practical aspects are now well known from a mono-site point of view. Nowadays, the growing of networks makes applications and informations more and more distributed and even mobile. Unfortunately, the tight relations between datas induced by the constraints prevent from easy partitioning of problems using existing frameworks. Still, there exist very few alternatives although it is crucial for constraint programming to handle these concerns.

This paper presents a formal framework for programming with constraints in a concurrent and explicitly distributed way, using Saraswat's concurrent constraint programming framework (`cc`, [Sar93]) as a basis. To deal with a really distributed computation, one has to abandon the global nature and uniqueness of the `cc` constraint set (called the *store*). Our calculus is thus based on the notion of *location*, which denotes a local store together with some processes acting on it. Locations have explicit addresses, can be dynamically created, and move during computations. Each location is opaque to others, except for those adjacent which are interfaced. It is very important because it ensures that local computation *are* local, and that global computation is *really* distributed. These minimal changes lead to a powerful distributed concurrent constraint framework. Then, we add a multiform timed construct which controls the execution of locations, and show how to write usual mechanisms, such as timeouts or exception handling.

The rest of the paper is organized as follows. The following section briefly presents the main concurrent constraint programming concepts. The third section presents the problem. The fourth section explains our extensions of the `cc` paradigm with distribution and mobility primitives. The fifth section details both syntax and operational semantics and discusses a few interesting features. The sixth section treats the addition of multiform time handling to our formalism. The seventh section is devoted to the syntax and semantics modifications. Then, we present some applications. At last, we describe related works, before to conclude.

2 Concurrent Constraint Programming

This work is based on the Concurrent Constraint Programming (`cc`) paradigm, a framework developed by Saraswat [Sar93] as a generalization of dataflow-, constraint logic-, and concurrent programming languages. Its main idea is to extend their control mechanisms by enabling the synchronization of processes on a set of constraints. Its flexibility makes it really appealing wrt other constraint formalisms.

2.1 Constraint Systems

Constraints enforce relationships among sets of variables. More precisely, each variable being associated with a certain set of values (its *domain*), constraints specify the sets of compatible values. All concurrent constraint languages are built over constraint systems [Sar93], which formalize these notions into a system of partial information, consisting of a set $\mathbf{C}$ of primitive constraints or tokens, closed under conjunction and existential quantification, and an entailment relation $\vdash$ that relates constraints and is reflexive and transitive. The set of constraints is the set of subsets of $\mathbf{C}$ closed under $\vdash$.

The generic notion of constraint system abstracts constraint problems. As a consequence, heterogenous constraint solvers can interact within the same framework. Moreover, it allows to encode constraint network consistency techniques and resolution strategies either as system built-ins or as user-written programs [RS93].

2.2 Constraint Stores

The state of a cc system, called its *store*, is determined by a conjunction of constraints (that is a constraint itself) which determines the partial information known about all variables. The store grows monotonically during the computation by addition of constraints. The basic cc model is based on the interaction of concurrent processes via the shared store, using four constructs :

- addition of a constraint to the store (*tell*),
- synchronization on the entailment of a given constraint by the store, then execution of some process (*ask*),
- hiding of variables,
- parallel execution of processes.

The denotation of a program is given by the set of stores in which execution can run to quiescence without adding more information.

It is to be noticed that in this paper we talk about cc processes whereas the cc terminology calls them agents, to prevent confusion with the multiagent meaning of this term, which generally involves notions of autonomy, intelligence and so on.

3 Distribution and cc

The main difficulty in distributing cc is to distribute the store. What can appear as a tautology is in fact a very hard problem, because the notion of shared store is precisely the core concept of classical cc.

3.1 What Belongs to Me Belongs to You

A solution might be to keep a conceptually shared view of the store at the user level, as in Oz [HvRS97], provided that the system has transparent mechanisms to handle distribution such as code and data copies or moves, automatic synchronizations and so on. This approach seems very appealing at first sight. As a matter of fact, even if network latencies and failures became negligible due to reliable communication protocols (which is rarely, if not never, the case), other pitfalls would prevent from adopting this solution. In particular, some parameters such as good load balancing, synchronization frequency and operators application order are essential for efficient resolutions but are very dependent of the constraint problem. Since there does not exist yet general methods to automatically optimize them, all the benefit waited from the distributed approach would be lost by such automatic badly tuned strategies.

3.2 What Belongs to Me Belongs to Me

Thus, a second solution is to let the user know that the store is distributed, and to provide him the means to properly handle distribution parameters. This gives the full control to the programmer, which is closer to the cc spirit. And as for consistency techniques in cc, this approach does not prevent from integrating frequently used distribution strategies at the system level as built-ins or libraries. But this also leads to reconsider the “all is shared” view of cc. Now, several stores are distributed among hosts, and processes are attached to a single one. They cannot directly interact each other unless they belong to the same location. In this case, only local communications are performed via the store as in classical `cc`. Moreover, since distribution strategies do not only involve data exchanges but also code moves between different hosts, processes should be able to travel from one store to an other. Therefore, distant communications are no longer performed by exchanges of passive datas but by moves of active processes.

4 A Distributed Concurrent Constraint Calculus

All operations of basic `cc` given Sect. 2 are allowed. In addition, a process can also define or call a procedure, or sequentially execute two operations. In the following sections, we summarize the key features of our calculus: locations, interfaces and mobility.

4.1 Locations

To distinguish the different stores were computations take place, each one has a different address. A store with its address is called *a location*. New locations are dynamically created by a *sublocation* operation. Since a creation is performed by a process acting in a given location, both locations are called *adjacent*.

4.2 Interfaces

As in basic `cc`, the processes can manipulate all variables and constraints of their own store, and can hide some variables to other processes of the same store. As security is an important concern for distributed calculi, data privacy must be enforced: one must also prevent unauthorized accesses from and to other locations. That is the role of interfaces, which are dedicated vectors of variables existing between any two adjacent stores. An *import interface* permits to see variables of a given adjacent store, while an *export interface* allows an adjacent store to see some internal variables.

4.3 Variables

The variables can be classified into several categories with an object oriented flavor. These categories also determine their accessibility from processes. This possibility to set different levels of security to read or write variable values is very interesting. Given a location:

- *local variables* are those of the store which are not hidden; they are readable and writable by all processes acting on the store;
- *hidden variables* are in the scope of an $\exists$; they are readable and writable in this scope and nowhere else (this is actually only syntactical, since these variables can be related to some public one by constraints, and then be affected by the modification of this public variable by any process);
- *protected variables* belong to adjacent locations but are viewable via their interface; they are only readable by all processes of the location.

4.4 Mobility

As explained before, we wish to enable code moves between locations. As a move can occur at any moment during an execution, it is meaningless to allow any single process to travel from one place to another without its environment. In the context of constraint programming, an environment means the concurrent processes and related store, which exactly corresponds to a location in our framework. Therefore, a location can only move as a whole, ie together with all its store, processes and sublocations. Moreover, it may only go on its own initiative to a given destination. Thus, two conditions are required:

- a *go* process must have been triggered at the location,
- at least one *accept* process must exist at the destination, and one of them must have allowed the moving location to be executed into the new location.

Note that this approach enables message-passing as well as messenger [FBDM97] styles of programming.

5 Formal Syntax and Semantics

In the following section, we present the formal syntax and explain in details its operational semantics.

5.1 Syntax

Let $\mathbf{D}$ be a set of primitive constraints, V an infinite set of variables, and $\mathcal{C} = \langle \mathbf{D}, \vdash, V, \{\exists_x \mid x \in V\}\rangle$ a constraint system built over $\mathbf{D}$ and V. We will denote $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$ the elements of $\mathbf{D}$, u, v, x, y, z those of V, and $\overline{x}$ a linear vector of variables. Given x, y two variables and $\overline{u}$, $\overline{v}$ two vectors of same dimension, $\{{}^{y}/_{x}\}$ denotes the substitution of x by y, and $\{{}^{\overline{v}}/_{\overline{u}}\}$ the substitution of each variable in $\overline{u}$ by the corresponding variable in $\overline{v}$.

The location names are built on an infinite alphabet Λ, whose elements are denoted α, β, γ. Let ε be the empty string and Φ, Ψ, Ω finite strings on Λ^*. In the following, we will often use Φ as a synonym for "the location whose name is Φ".

The processes are combinations of basic constructs given in Tab.1. As usual, parentheses can be used for grouping.

$P ::=$	$\mathbf{c}$	tell
$\mid$	$\mathbf{c} \rightarrow P$	ask
$\mid$	$\exists x.P$	hidden variable
$\mid$	$P; P$	sequence
$\mid$	$P \mid P$	parallel composition
$\mid$	0	nil process
$\mid$	$\alpha(\overline{x})(\overline{y})[P]$	sublocation
$\mid$	$@\ \Phi$	go
$\mid$	$\triangleright\alpha(\overline{x})(\overline{y})$	accept
$\mid$	$p(\overline{x}) :: P$	procedure definition
$\mid$	$p(\overline{x})$	procedure call

Table1. Syntax

5.2 Semantics

The operational semantics has similarities with the *Distributed Chemical Abstract Machine* formalism [FGL+96]. The processes can be thought as molecules floating in solutions and interacting by chemical reactions, by means of structural and reduction rules.

Solutions. Let $\Phi \in \Lambda^*$ be a location, then (the subscripted name of the location can be omitted if there is no ambiguity):

- $\mathcal{S}_\Phi$ is the constraint store of Φ,
- $\mathcal{A}_\Phi$ is the set of local active processes,
- Def_Φ is the set of procedure definitions available in Φ,
- Adj_Φ is the set of the names of locations adjacent to Φ,
- $\overline{In}_\Phi$ is the vector of import interface variables,
- $\overline{Out}_\Phi$ is the vector of export interface variables.

A solution represents the state of a given location: Given $\Phi \in \Lambda^*$, the corresponding solution is the tuple $\langle \mathcal{A}_\Phi, \mathcal{S}_\Phi \rangle$. For the sake of readability, other sets and vectors previously defined are implicitly associated with their respective location, but do not appear in solutions.

As several locations are generally involved in computation, a global state of the abstract machine is compound of several solutions separated by $\|$. A composition of rules models an evolution of these states, that is a distributed concurrent constraint computation. Again, for clarity, only solutions that take part in reactions appear in rules.

Structural rules. They are used for syntactical rearrangements of terms. They are reversible and expressed using the $\rightharpoonup$ (*heating*) and $\leftharpoondown$ (*cooling*) constructs, such that $S_1 \rightharpoonup S_2$ is the inverse of $S_1 \leftharpoondown S_2$ (the latter being read from right to left). Whenever it is possible, both rules are packed into a single one, $S_1 \rightleftharpoons S_2$. **Sublocation (1)** and **Sublocation (2)** are two such rules.

Reduction rules. They are of the form $S_1 \rightarrow S_2$ and model actual computation steps. They are not reversible. For example, **Tell** is a reduction rule.

5.3 Stores and Locations

A location semantically corresponds to a solution. It is created by giving

- a new relative name (in our formalism, an element of Λ, supposed unique): if the creation process is performed in Φ, and α is the given name, then $\Phi\alpha$ denotes the new sublocation,
- an initial process to execute (note that the store is initially empty),
- import and export interfaces: as said before, data privacy is enforced by means of the interfaces.

Each location Φ stores the names of its adjacent locations in Adj_Φ. Thus, Adj_Φ is augmented with $\Phi\alpha$ and $Adj_{\Phi\alpha}$ is initialized with Φ. Moreover, a new location inherits of the procedures defined in its superlocation. In our example, $Def_{\Phi\alpha}$ is set to Def_Φ.

$$\textsc{Sublocation (1)}$$
$$\langle \mathcal{A} \cup \{\alpha(\overline{x})(\overline{y})\,[P]\}\,, \mathcal{S}_{\Phi}\rangle \rightharpoonup \langle \mathcal{A}, \mathcal{S}_{\Phi}\rangle \parallel \langle \{P\}\,, \mathcal{S}_{\Phi\alpha}\rangle$$

and $\mathcal{S}_{\Phi\alpha} := \top$
$\overline{In}_{\Phi\alpha} := \overline{x}$
$\overline{Out}_{\Phi\alpha} := \overline{y}$
$Adj_{\Phi} := Adj_{\Phi} \cup \{\Phi\alpha\}$
$Adj_{\Phi\alpha} := \{\Phi\}$
$Def_{\Phi\alpha} := Def_{\Phi}$

5.4 Basic Constructs

Once a store has been created, the basic `cc` constructs can be performed.

Tell. A constraint can be added to the local store. Note that while there is no consistency checking , a store may become inconsistent. But unlike in other `cc` paradigms, it is only harmful for adjacent stores, if the inconsistency affects their interfaces. Thus, an appropriate choice of interfaces prevents insidious viruses-like locations to affect their host.

$$\textsc{Tell}$$
$$\langle \mathcal{A} \cup \{\mathbf{c}\}\,, \mathcal{S}_{\Phi}\rangle \to \langle \mathcal{A}, \mathcal{S}_{\Phi} \wedge \mathbf{c}\rangle$$

Ask. A process can be triggered when a constraint is entailed by the conjunction of the local store ($\mathcal{S}_{\Phi}$) and all the viewable parts of adjacent stores ($\bigwedge_{\alpha\in Adj_{\Phi}}\left(\mathcal{S}_{\alpha}|_{\overline{Out}_{\Phi}}\right)$), or removed if the negation of the constraint is entailed. *Ask* suspends until one of these two conditions is reached.

$$\textsc{Ask}$$
$$\frac{\mathcal{S}_{\Phi} \wedge \bigwedge_{\alpha\in Adj_{\Phi}}\left(\mathcal{S}_{\alpha}|_{\overline{Out}_{\Phi}}\right) \vdash \mathbf{c}}{\langle \mathcal{A} \cup \{\mathbf{c} \to P\}\,, \mathcal{S}_{\Phi}\rangle \to \langle \mathcal{A} \cup \{P\}\,, \mathcal{S}_{\Phi}\rangle}$$

$$\frac{\mathcal{S}_{\Phi} \wedge \bigwedge_{\alpha\in Adj_{\Phi}}\left(\mathcal{S}_{\alpha}|_{\overline{Out}_{\Phi}}\right) \vdash \neg\mathbf{c}}{\langle \mathcal{A} \cup \{\mathbf{c} \to P\}\,, \mathcal{S}_{\Phi}\rangle \to \qquad \langle \mathcal{A}, \mathcal{S}_{\Phi}\rangle}$$

Hiding. A process can hide a variable. This is essentially syntactical, since a constraint can relate such variable to a public one such that each modification of the latter affects the former.

$$\textsc{Hidden variable}$$
$$\langle \mathcal{A} \cup \{\exists x.P\}\,, \mathcal{S}_{\Phi}\rangle \rightleftharpoons \langle \mathcal{A} \cup \{P\,[{}^{y}/{}_{x}]\}\,, \mathcal{S}_{\Phi}\rangle$$
where y is fresh

Execution ordering. Processes can be executed sequentially or concurrently.

$$\textsc{Sequence}$$

$$\frac{\langle\{P_1\},\mathcal{S}_\Phi\rangle \quad \overset{\rightarrow}{\rightleftharpoons} \quad \langle\{P_1'\},\mathcal{S}_\Phi'\rangle}{\langle\mathcal{A}\cup\{P_1;P_2\},\mathcal{S}_\Phi\rangle \quad \rightarrow \langle\mathcal{A}\cup\{P_1';P_2\},\mathcal{S}_\Phi'\rangle}$$

$$\frac{\langle\{P_1,P_3\},\mathcal{S}_\Phi\rangle \quad \overset{\rightarrow}{\rightleftharpoons} \quad \langle\{P'\},\mathcal{S}_\Phi'\rangle}{\langle\mathcal{A}\cup\{(P_1;P_2),P_3\},\mathcal{S}_\Phi\rangle \rightarrow \langle\mathcal{A}\cup\{P';P_2\},\mathcal{S}_\Phi'\rangle}$$

$$\frac{\langle\{P_1\},\mathcal{S}_\Phi\rangle \quad \overset{\rightarrow}{\rightleftharpoons} \quad \langle\{\},\mathcal{S}_\Phi'\rangle}{\langle\mathcal{A}\cup\{P_1;P_2\},\mathcal{S}_\Phi\rangle \quad \rightarrow \quad \langle\mathcal{A}\cup\{P_2\},\mathcal{S}_\Phi'\rangle}$$

$$\textsc{Parallel composition}$$

$$\langle\mathcal{A}\cup\{P_1 \mid P_2\},\mathcal{S}_\Phi\rangle \quad \rightleftharpoons \langle\mathcal{A}\cup\{P_1,P_2\},\mathcal{S}_\Phi\rangle$$

Nil. The 0 process does nothing.

$$\textsc{Nil}$$

$$\langle\mathcal{A}\cup\{0\},\mathcal{S}_\Phi\rangle \rightleftharpoons \langle\mathcal{A},\mathcal{S}_\Phi\rangle$$

5.5 Procedures

For a matter of simplicity, we assume that there are not procedures with same name and different arities. This can be achieved without loss of generality by replacing each name p with arity n by the name p/n. Each procedure defined at the location Φ is stored in the dedicated Def_Φ set. If p is already defined, then its definition is replaced by the new one. A procedure call simply replaces the head by the body with substituted variables.

$$\textsc{Procedure definition}$$

$$\langle\mathcal{A}\cup\{p(\overline{y}) :: P\},\mathcal{S}_\Phi\rangle \rightleftharpoons \qquad \langle\mathcal{A},\mathcal{S}_\Phi\rangle$$

$$\text{and } Def_\Phi := (Def_\Phi \setminus \{p(\overline{z}) :: Q\}) \cup \{p(\overline{y}) :: P\}$$

$$\textsc{Procedure call}$$

$$\langle\mathcal{A}\cup\{p(\overline{x})\},\mathcal{S}_\Phi\rangle \quad \rightarrow \left\langle\mathcal{A}\cup\left\{P\left\{{}^{\overline{y}}/_{\overline{x}}\right\}\right\},\mathcal{S}_\Phi\right\rangle$$

$$\text{where } p(\overline{y}) :: P \in Def_\Phi$$

5.6 Mobility

A location move is decomposed into three phases. First, when a *go* is performed upon a location, all sublocations are recursively frozen using the cooling part of the structural rules. For this matter we need a cooling version of the **Sublocation (1)** rule.

Sublocation (2)

$$\frac{\langle\{P'\},\mathcal{S}_{\Phi\alpha}\rangle \quad \leftharpoondown^{*} \langle\{P\},\mathcal{S}_{\Phi\alpha}\rangle \parallel \langle\mathcal{A}_{\delta 1},\mathcal{S}_{\delta 1}\rangle \parallel \ldots \parallel \langle\mathcal{A}_{\delta n},\mathcal{S}_{\delta n}\rangle}{\left\langle\mathcal{A}_{\Phi}\cup\left\{\alpha(\overline{In_\alpha})(\overline{Out_\alpha})\left[\mathcal{S}_{\Phi\alpha};P'\right]\right\},\mathcal{S}_{\Phi}\right\rangle \leftharpoondown \quad \langle\mathcal{A}_{\Phi},\mathcal{S}_{\Phi}\rangle \parallel \langle\{P\},\mathcal{S}_{\Phi\alpha}\rangle}$$

$$\begin{array}{ll} \text{and} & Adj_{\Phi} := Adj_{\Phi}\setminus\{\Phi\alpha\} \\ \text{where} & Adj_{\Phi\alpha} = \{\Phi,\delta_1,\ldots,\delta_n\} \\ & P' = P \mid \delta_1(\overline{In_{\delta 1}})(\overline{Out_{\delta 1}})\left[P_{\delta 1}\right] \mid \ldots \mid \delta_n(\overline{In_{\delta n}})(\overline{Out_{\delta n}})\left[P_{\delta n}\right] \end{array}$$

As said before, the *go* operation is performed by a process attached to the location-to-move itself, and not by an external entity. Then, the frozen location is routed to its destination, where it is left in a waiting state until its treatment.

Go

$$\frac{\langle\mathcal{A}_{\Phi}\cup\{\alpha(\overline{In_\alpha})(\overline{Out_\alpha})[P]\},\mathcal{S}_{\Phi}\rangle \quad \leftharpoondown^{*} \quad \langle\mathcal{A}_{\Phi\alpha},\mathcal{S}_{\Phi\alpha}\rangle \parallel \langle\mathcal{A}_{\Phi},\mathcal{S}_{\Phi}\rangle}{\langle\mathcal{A}_{\Phi\alpha}\cup\{@\ \Omega\},\mathcal{S}_{\Phi\alpha}\rangle \parallel \langle\mathcal{A}_{\Phi},\mathcal{S}_{\Phi}\rangle \parallel \langle\mathcal{A}_{\Omega},\mathcal{S}_{\Omega}\rangle \rightarrow \left\langle\mathcal{A}_{\Omega}\cup\left\{\Box\alpha(\overline{In_\alpha})(\overline{Out_\alpha})[P]\right\},\mathcal{S}_{\Omega}\right\rangle \parallel \langle\mathcal{A}_{\Phi},\mathcal{S}_{\Phi}\rangle}$$

Finally, the moving location can be unfrozen if and only if there exists an *accept* process of the host which allows its execution. Each location is free to execute as many *accept* as wanted (including none which means that all foreign locations are refused).

Accept

$$\langle\mathcal{A}\cup\{\triangleright\beta(\overline{x})(\overline{y}),\Box\alpha(\overline{u}_\alpha)(\overline{v}_\alpha)\left[P\right]\},\mathcal{S}_{\Phi}\rangle \rightarrow \left\langle\mathcal{A}\left[{}^{\alpha}/_{\beta}\right]\cup\left\{\alpha(\overline{x})(\overline{y})\left[P\left[{}^{\overline{x}}/_{\overline{u}}\right]\left[{}^{\overline{y}}/_{\overline{v}}\right]\right]\right\},\mathcal{S}_{\Phi}\right\rangle$$

5.7 Some Security Concerns

Secure data accesses are very important in a distributed context. Let now discuss a few commonly used mechanisms from our language point of view.

Privacy. As pointed out Sec. 4.3, variables can be classified into local, hidden and protected variables. Moreover, the two last categories can be combined to eliminate the problem of hidden variables possibly indirectly modified: *private variables* belong to a dedicated adjacent location and appear in its interface, which is hidden by an $\exists$. Thus these variables are only readable by processes in the scope of $\exists$ and cannot be directly or indirectly modified, except by the move of a process, if it is accepted by the private location.

Finally, considering the following fragment of code, w is local, x is hidden, u and v are protected, z is private:

$$\alpha(u,v)(w)[\rho()(z)[\ldots] \mid \mathbf{c}(u) \rightarrow \exists x.\mathbf{d}(x,w)\ldots]$$

Authentication. Authentication of mobile processes is often required in distributed computations. Imagine for example a secure database server. Some prospecting agents travel from one place to another to collect new datas and can add or remove records when they go back to the server. On the other hand, clients can only connect to consult the database. Thus it is necessary to distinguish the ones from the others, which is very easy using interfaces and accept: for example, just test if a given constraint holds, as the equality of a key kept by the server and only known by the agents with the value of a variable of the arriving locations.

Sandboxing. Accept implicitly implies a kind of sandboxing because executions of arriving locations are always restricted to their own store, thus a good choice of interface permits the isolation of unsecure mobile locations.

Firewalls. It is also very easy to write a firewall. If a given location wants to keep its name secret, it does not accept any arriving location and creates a location, with the appropriate interfaces (the variables which can be read from the outside and those that it can modify), which accepts connections on its behalf.

6 Multiform Time

6.1 Timed Systems and cc

As pointed out by Berry [Ber93], timed concurrent systems deal with processes that coordinate with each other. In particular, one should be able to start or stop any process at any time, where *time* is not produced by a physical clock but is a logical notion. Moreover, it should be multiform, that is generated by the repetition of any signal, depending on what one wants to encode: "abort after 30 meters" or "start after 10 seconds".

As the constraint accumulation is monotone, several additions of the same constraint have no effect on the store, thus *tell* does not suffice to model signal repetition. We have chosen to model signals with the modifications of the projection of the store on given variables. We will discuss Sec. 9 two alternative solutions. In the first hand, we think our approach has the main advantage to allow a more precise control on the actual delay between ticks. In the other hand, this characterization permits a good formalization of the calculus. But it is unconstrained enough to be in practice adapted to the situation. The appealing flexibility inherent to cc is back: in function of the desired granularity, a "store modification" can vary from the syntactical level to the deepest low-level of implementation. To illustrate this idea, consider a store $\mathcal{S} = \{x = 1, y > 0\}$ where the constraint $\mathbf{c} = x > y$ is added. Let observe the modification of $\mathcal{S}|_x$ and $\mathcal{S}|_y$. From a syntactical point of view, both are modified since they contain $\mathbf{c}$ which concerns x and y. But from a lower level, where variable domains modifications are taken into account, only $\mathcal{S}|_y$ is affected, since the addition of $\mathbf{c}$ only reduces the domain of y.

6.2 Coordination and Preemption

The simplest way to think about coordination is to consider it as a side effect of communication. In classical cc, it is performed via available informations constituting the constraints store and the *ask* construct. But it is not sufficient to our needs. Firstly, it cannot model the *process preemption* that is the control of processes life and death. The *ask* operation is not sufficient to model a situation where a process should not know why and when it gets preempted. Secondly, this scheme works only for processes, while we are interested in the control of locations.

The calculus presented above is based on *autonomous* locations: in particular, they can move to another site on their own initiative. But this freedom can become a nightmare for the host which has no mean to control the activity of the locations it accepts: a bad intended guest could for example block all others by an intensive activity (this is the principle of *Denial of Service* attacks on Internet servers). Therefore we introduce now a notion of hierarchy between locations : let α be a location, it *has authority* on every location created or accepted by one of its local processes. Conversely, all these locations are *submitted* to α. This means that α has the power to trigger and suspend their activity.

7 A Timed Calculus

7.1 Syntax

Time awareness is added to our language with a single new construct, called *trap*. It can be seen as a timeout construct, or as a kind of exception-like mechanism (these two points will be discussed later): given a vector of variables $\overline{x}$ and two submitted stores α and β, $\overline{x}\dagger(\alpha, \beta)$ is the process which suspends α and starts β when the store is modified wrt these variables. We will call such variables *watched*, while related locations α and β are respectively said *trapped by* and *waiting for* the variables.

7.2 Semantics

All rules detailed Sec. 5.2 are part of the new semantics unless they are replaced. In addition, to handle traps, one must modify the preceding semantics to be able:

- to detect a store modification, to stop or start related locations,
- to remind which locations are concerned.

In the following, we mention for each new rule if it replaces an older one or if it is added to the semantics.

Detection of store modifications. Modifications can only occur when constraints are added to the store, that is during a *tell*. Each time such operation is performed, if the store is modified, one needs to detect the variables involved. As explained before, the level of the detection may vary. Thus, when in the following we use the generic relation $\neq$, remind that in practice it is tailored to the needs of the programmer.

Definition 1. *Given a location Φ and a constraint $\mathbf{c}$, the set of all variables touched by the addition of $\mathbf{c}$ to $\mathcal{S}_\Phi$ is*

$$N_\Phi(\mathbf{c}) = \{x \,|(\mathcal{S}_\Phi \cup \mathbf{c})|_x \neq \mathcal{S}_\Phi|_x\}$$

The locations *trapped by* and *waiting for* these variables are respectively frozen or awoken.

Definition 2. *Let define $\mathcal{T}_\Phi$ and $\mathcal{W}_\Phi$ the sets of the trapped and waiting locations names in Φ. Each location name is superscripted with the name of a variable it is related to.*

Definition 3. *Given a variable x, the set of all submitted locations trapped by x in Φ is*

$$\mathcal{T}_\Phi^x = \{\alpha|\exists\alpha^x \in \mathcal{T}_\Phi\}$$

Definition 4. *Given a variable x, the set of all submitted locations waiting for x in Φ is*

$$\mathcal{W}_\Phi^x = \{\alpha|\exists\alpha^x \in \mathcal{W}_\Phi\}$$

The *trap* construct. The *trap* construct gives means to control the execution of submitted sublocations. $\mathcal{T}_\Phi$ and $\mathcal{W}_\Phi$ are respectively augmented with the names of the new trapped and waiting locations, each one appearing superscripted with each name of the involved variables.

$$\textsc{Trap}\ \text{(added rule)}$$
$$\langle \mathcal{A}_\Phi \cup \{\overline{x}\dagger(\alpha,\beta)\}, \mathcal{S}_\Phi\rangle \rightarrow \langle \mathcal{A}_\Phi, \mathcal{S}_\Phi\rangle$$
$$\text{and} \qquad \mathcal{T}_\Phi := \mathcal{T}_\Phi \cup \bigcup_{y\in\overline{x}} \{\alpha^y\}$$
$$\mathcal{W}_\Phi := \mathcal{W}_\Phi \cup \bigcup_{y\in\overline{x}} \{\beta^y\}$$

Then trapped and waiting locations can be interrupted or triggered by the addition of $\mathbf{c}$ to S.

$$\textsc{Tell}\ \text{(replaces the former } \textsc{Tell} \text{ rule)}$$
$$\frac{\langle \mathcal{A}', \mathcal{S}_\Phi\rangle \qquad \leftharpoondown^* \qquad \langle \mathcal{A}, \mathcal{S}_\Phi\rangle \parallel \langle \mathcal{A}_{\alpha 1}, \mathcal{S}_{\alpha 1}\rangle \parallel \ldots \parallel \langle \mathcal{A}_{\alpha n}, \mathcal{S}_{\alpha n}\rangle}{\langle \mathcal{A} \cup \{\mathbf{c}, \Diamond\beta_1(\overline{u}_1)(\overline{v}_1)\,[Q_1], \ldots, \Diamond\beta_m(\overline{u}_m)(\overline{v}_m)\,[Q_m]\}, \mathcal{S}_\Phi\rangle \rightarrow \langle \mathcal{A}', \mathcal{S}_\Phi \wedge \mathbf{c}\rangle \parallel \langle \mathcal{A}_{\beta 1}, \mathcal{S}_{\beta 1}\rangle \parallel \ldots \parallel \langle \mathcal{A}_{\beta m}, \mathcal{S}_{\beta m}\rangle}$$

$$\begin{array}{lll}
\text{and} \quad \mathcal{A}' & = & \mathcal{A} \cup \{\Diamond\alpha_1(\overline{x}_1)(\overline{y}_1)[P_1], \ldots, \Diamond\alpha_n(\overline{x}_n)(\overline{y}_n)[P_n]\} \\
\forall i \in [1..m] & & \langle \mathcal{A}_{\beta i}, S_{\beta i}\rangle = \langle \{Q_i\}, \top\rangle \\
\mathcal{T}_\Phi & := & \mathcal{T}_\Phi \setminus \bigcup_{y\in N_\Phi(c)} \mathcal{T}_\Phi^y \quad \text{where} \quad \bigcup_{y\in N_\Phi(c)} \mathcal{T}_\Phi^y = \{\alpha_1, \ldots, \alpha_n\} \\
\mathcal{W}_\Phi & := & \mathcal{W}_\Phi \setminus \bigcup_{y\in N_\Phi(c)} \mathcal{W}_\Phi^y \quad \text{where} \quad \bigcup_{y\in N_\Phi(c)} \mathcal{W}_\Phi^y = \{\beta_1, \ldots, \beta_m\}
\end{array}$$

This new **Tell** rule may appear a bit complicated. In fact, it only states that when a constraint $\mathbf{c}$ is added to the store, this one may be modified wrt some variables. $N_\Phi(c)$ is the set of such variables. A number of submitted locations were waiting for these modifications, either to be awoken (if their names appear superscripted with one of the variables in $\mathcal{W}_\Phi$) or to be suspended (in $\mathcal{T}_\Phi$). So, the former (the β_i) are triggered, while the latter (the α_j) are frozen.

Control on new submitted locations. At last, to improve control on new submitted locations, one must ensure that they cannot start their execution before being triggered by their host. Two cases are to be handled: new locations created with the **Sublocation (1)** rule, and newly arrived ones, accepted using the **Accept** rule. As the latter only prepares newly arrived locations to be reduced by the former rule, it suffices to modify this one to handle both cases. Thus, it is modified in such a way that all the new submitted locations are now frozen.

$$\textsc{Sublocation (1)}\ \text{(replaces the former } \textsc{Sublocation (1)} \text{ rule)}$$
$$\langle \mathcal{A} \cup \{\alpha(\overline{x})(\overline{y})\,[P]\}, \mathcal{S}_\Phi\rangle \rightharpoonup \langle \mathcal{A} \cup \{\Diamond\alpha(\overline{x})(\overline{y})\,[P]\}, \mathcal{S}_\Phi\rangle$$
$$\begin{array}{l}
\text{and } \overline{In}_{\Phi\alpha} := \overline{x} \\
\quad \overline{Out}_{\Phi\alpha} := \overline{y} \\
\quad Adj_\Phi := Adj_\Phi \cup \{\Phi\alpha\} \\
\quad Adj_{\Phi\alpha} := \{\Phi\} \\
\quad Def_{\Phi\alpha} := Def_\Phi
\end{array}$$

8 Applications

The trap construct is simple enough to make semantics extensions straightforward, but it is very powerful. We will see in the following several useful constructs which can be derived from it. An auxiliary location may be necessary: we suppose that ν, defined by $\nu()()[]$, is a location submitted to the one where the examples are running.

8.1 Triggering of a process

The simplest construct we can obtain using trap consists in starting the execution of a location α when a variable of $\overline{x}$ is touched. We can define this triggering construct *next* as

$$next\,\overline{x}\,do\,\alpha \;\equiv\; \overline{x}\dagger(\nu, \alpha)$$

8.2 Suspension of a process

The dual construct is when $\overline{x}$ $suspend\,\alpha$ which stops α when a modification affects $\overline{x}$.

$$when\,\overline{x}\,suspend\,\alpha \;\equiv\; \overline{x}\dagger(\alpha, \nu)$$

It is to be noticed that due to the trap semantics, we can only define the suspension. An extension would be necessary to model the abort, which would kill α.

8.3 Exceptions

An exception-like mechanism can also be defined: let $try\,e\,do\,\alpha\,else\,\beta$ be the process starting α and stopping α to start β if the exception e is raised by α. First, one needs to be able to detect such rise: as it occurs in α to be handle in the location α is submitted to, it must be seen through the output interface of α. Thus, the *try* construct is defined as follows, where α has been defined as $\alpha(\ldots)(\ldots, e_\alpha)[\ldots]$:

$$try\,e\,do\,\alpha\,else\,\beta \equiv \exists e.(e = e_\alpha; e\dagger(\alpha, \beta))$$

8.4 Timeouts

suspend allows to halt a location when a signal s is emitted (that is a modification of the store involves the variable s). But sometimes, it is preferable to wait for a moment (ie a certain number of signal emissions) before to suspend the location. $clock\,s\,do\,\alpha\,timeout\,\mathrm{n}$ waits n emissions of s then stops α:

$$\begin{array}{l} clock\,s\,do\,\alpha\,timeout\,\mathrm{n} \equiv \\ \exists g.(\beta()(g)[(touch(g)] \\ \quad ;\underbrace{next\,s\,do\,next\,s\,do\,\ldots next\,s\,do}_{\mathrm{n\ times}}\,\beta \\ \quad \mid when\,g\,suspend\,\alpha) \end{array}$$

Here we suppose that *touch* modifies the store wrt the given variable. β sleeps waiting for n emissions of s, then triggers α by modifying g.

8.5 Temporization

In the same manner, one can define the process which waits n emissions of s then triggers α:

$$\begin{aligned}&clock\, s\, do\, \alpha\, after\, \mathrm{n} \equiv\\ &\underbrace{next\, s\, do\, next\, s\, do\, \ldots next\, s\, do}_{\text{n times}}\, \alpha\end{aligned}$$

9 Related Works

Distribution has already been studied in a concurrent constraint programming context. Moreover, as said Sec. 6, several approaches have influenced our work on multiform time.

9.1 `cc` and distribution

`Oz` [HvRS97] (implemented in the `Mozart` system) maintains a view of the common store across multiple sites by complex built-in mechanisms. In this manner, the network is abstracted away and shared datas are automatically updated, duplicated, moved or synchronized when and where needed.

Saraswat's family of distributed languages (`dc` [Sar93], `Janus` [SKL90]) imposes restrictions on syntax and domain of computation to ensure that the store can effectively be decomposed into distinct parts. Roughly, the two key concepts are that all non ground variables are to be viewed as point-to-point directional communication channels (thus only one process is allowed to tell some constraint on a given variable), and that syntactically correct programs should not cause the store to be inconsistent at run-time.

Réty's merge of `cc` and π-calculus [Rét98] imposes less restrictive conditions on the store to be distributed (parts of the store are sets of constraints only related to each other set by equality constraints on communication channels), and extends `cc` with several features, firstly channels and the *send* and *receive* message constructs borrowed from the π-calculus, secondly store operators to manipulate any part of the store as a whole, and lastly the *link* primitive which allows to link together distributed sets of variables.

To our point of view, the main drawbacks of these approaches are either the automatization is not satisfactory because it does not allow enough fine distribution control to the programmer, or the restrictions are too strong, particularly for dynamic distributed computations. In definitive, the shared store is never really questioned back, which does not allow dynamic large scaled distributed computations.

9.2 The Berry preemption model

Berry [Ber93] treats time-dependent and particularly reactive systems driven with signals emissions and receptions. Time is *multiform*, that is generated by the repetition of any signal. For him, there is an intrinsic contradiction between classical time-independent languages and time accuracy. Moreover, there should not exist restrictions on preemption primitives use and in particular a process P should not know when and why it gets preempted. As a consequence, he insists on the necessity to work with a time-dependent model to correctly handle time. His formalism is based on the *zero delay hypothesis*, also called *perfect synchrony*, which states that any process receives immediately what it emits. Through a parallel with `Unix`, he focuses on the notion of process preemption and distinguishes abortion (`^C`) and suspension (`^Z;fg`). He defines several variants of each and in particular *weak* (or *may*) and *strong* (or *must*) primitives of each sort, which are respectively effective immediately or at next step. Our *trap* operation is inspired from these constructs. This framework is the basis of the `Esterel` language.

9.3 `cc` and time

The main integrations of time into `cc` are `Timed cc` (in short `tcc`) and its derivatives from Saraswat et al. [GJS96] and *tccp* from de Boer and Gabbrielli [dBG95]. Both deal with the hypothesis of *bounded asynchrony*, an adaptation of the *perfect synchrony* hypothesis of synchronous languages [Ber93], without its inherent temporal paradoxes (for example, to require a signal to be present iff it is not present) by supposing that computation takes a bounded period of time rather than being instantaneous.

Saraswat et al. have proposed several models, from discrete timed `tcc` to continuous timed with default handling `Hybrid cc`, all based on the principle that a system evolves in cycles corresponding to time intervals, where each time interval is determined by the time needed for all active processes to reach quiescence. The programmer has to transfer explicitly the information from a time instant to the next and to install future running processes before the next cycle begins. Special temporal primitives are added to the standard ones for this purpose.

On the other hand, *tccp* consider only discrete time where one unit is the time needed by the underlying constraint system to handle one *tell* or *ask* primitive. This is simpler than the previous (discrete) models, for semantics as well as for programming because explicit transfer of information between phases is no longer required.

A major common default of these two frameworks is that time intervals depend on undeterminate parameters (duration to reach quiescence or to perform a primitive) which can vary in an uncontrolled way.

10 Conclusion

This work extends Saraswat concurrent constraint paradigm to handle distributed computations and time. A calculus with formal syntax and operational semantics has been defined. This is a work in progress which is far from being achieved. We have now several perspectives to explore. It would be interesting to study the extension of an actual constraint programming language within our framework. We also think that this formalism would be a good basis for a multiagent programming language. We now concentrate our efforts on these two subjects.

Acknowledgments. This work benefited greatly from discussions with Philippe Codognet. The author wishes to thank him for his fruitful ideas and comments.

References

[Ber93] G. Berry. Preemption in concurrent systems. In Rudrapatna K. Shyamasundar, editor, *Proceedings of Foundations of Software Technology and Theoretical Computer Science*, volume 761 of *LNCS*, pages 72–93, Berlin, Germany, December 1993. Springer.

[dBG95] Frank S. de Boer and Maurizio Gabbrielli. Modelling real-time in concurrent constraint programming. In John Lloyd, editor, *Proceedings of the International Symposium on Logic Programming*, pages 528–542, Cambridge, December 4–7 1995. MIT Press.

[FBDM97] Munehiro Fukuda, Lubomir F. Bic, Michael B. Dillencourt, and Fehima Merchant. Messages versus messengers in distributed programming. In *Proceedings of the 17th International Conference on Distributed Computing Systems (ICDCS)*, Baltimore, Md., May 1997.

[FGL+96] Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile processes. In Ugo Montanari and Vladimiro Sassone, editors, *CONCUR '96: Concurrency Theory, 7th International Conference*, volume 1119 of *Lecture Notes in Computer Science*, pages 406–421, Pisa, Italy, 26–29 August 1996. Springer-Verlag.

[GJS96] Vineet Gupta, Radha Jagadeesan, and Vijay Saraswat. Models for concurrent constraint programming. In Ugo Montanari and Vladimiro Sassone, editors, *CONCUR '96: Concurrency Theory, 7th International Conference*, volume 1119 of *Lecture Notes in Computer Science*, pages 66–83, Pisa, Italy, 26–29 August 1996. Springer-Verlag.

[HvRS97] Seif Haridi, Peter van Roy, and Gert Smolka. An overview of the design of Distributed Oz. In ACM, editor, *PASCO '97. Proceedings of the second international symposium on parallel symbolic computation, July 20–22, 1997, Maui, HI*, pages 176–187, New York, NY 10036, USA, 1997. ACM Press.

[RS93] Francesca Rossi and Vijay A. Saraswat. Constraint satisfaction and local consistency in concurrent constraint logic programming. Presented at NATO ASI on Constraint Programming, Pärnu, Estonia, Aug 1993.

[Rét98] Jean-Hugues Réty. Distributed concurrent constraint programming. *Fundamenta Informaticae*, 34(3):323–346, June 1998.

[Sar93] Vijay A. Saraswat. *Concurrent Constraint Programming.* ACM Doctoral Dissertation Awards: Logic Programming. The MIT Press, Cambridge, MA, 1993.

[SKL90] Vijay A. Saraswat, Ken Kahn, and Jacob Levy. Janus: A step towards distributed constraint programming. In Manuel Debray, Saumya; Hermenegildo, editor, *Proceedings of the 1990 North American Conference on Logic Programming*, pages 421–438, Austin, TX, October 1990. MIT Press.