Self-assembling Trees

Vincent Danos[1] ,

Équipe PPS, CNRS & Université Paris VII

Jean Krivine[2] ,

INRIA Rocquencourt & Université Paris VI

Fabien Tarissan[3]

Équipe PPS, CNRS & Université Paris VII

Abstract

RCCS is a variant of Milner's CCS where processes are allowed a controlled form of backtracking. It turns out that the RCCS reinterpretation of a CCS process is equivalent, in the sense of weak bisimilarity, to its causal transition system in CCS. This can be used to develop an efficient method for designing distributed algorithms, which we illustrate here by deriving a distributed algorithm for assembling trees. Such a problem requires solving a highly distributed consensus, and a comparison with a traditional CCS-based solution shows that the code we obtain is shorter, easier to understand, and easier to prove correct by hand, or even to verify.

Keywords: Self Assembling, RCCS, CCS, Distributed Consensus.

1 Introduction

We propose in this paper to illustrate a method for deriving distributed algorithms. The broad idea is to solve a simpler problem, and then reinterpret the obtained solution assuming a generic distributed backtracking mechanism. This is reminiscent of the classic breakdown of solutions to NP problems into an exploration (guessing the solution) and a verification phase (checking the guess is correct). It is also reminiscent of simulated annealing methods where a locally-driven search is backed by a random perturbation. Another analogy is with declarative programming where terse solutions can be obtained because the ambient evaluation framework includes a generic enumeration mechanism.

[1] Email: vincent.danos@pps.jussieu.fr
[2] Email: jean.krivine@inria.fr
[3] Email: fabien.tarissan@pps.jussieu.fr

This paper is electronically published in
Electronic Notes in Theoretical Computer Science

It turns out that the notion of a solution to a simpler problem can be neatly characterised in terms of the theory of concurrent systems, using the notion of causal transition system, and so does the correctness of the generic backtracking mechanism. A rather general result then ensures that the reinterpreted solution is indeed a solution to the original problem [4].

This compares best with direct approaches when the problem of interest needs reaching a consensus which is itself highly distributed. Thus, for the purpose of illustrating the method, we choose a class of problems which is a simple idealisation of the phenomenon of self-assembly, where simple parts assemble in some predefined spatial arrangement by means of local and asynchronous interactions. Solutions of such problems indeed involve arbitrarily complex distributed consensus.

Specifically, we derive a distributed algorithm for an ensemble of processes to self-assemble in patterns described as trees. To formulate the algorithm, we use a partially reversible derivative of CCS [12], called RCCS, which introduces a distinction between reversible and irreversible computation steps, together with a notion of distributed memory which allows backtracking reversible steps [3].

The algorithm itself is obtained indirectly. One first defines a simple CCS algorithm such that any allowed tree construction can be simulated, and conversely all trees resulting from a series of local interactions are allowed. This is not yet a solution since the induced assembly may deadlock, but it gets very close to being one. Indeed, by merely reinterpreting the same algorithm in RCCS, and thus allowing backtrack on reversible actions, one obtains a real solution. For the sake of evaluating the method we compare the first algorithm with a direct solution in CCS which explicitly copes with deadlocks. One sees clearly that the latter is both harder to understand, and to prove correct, and also assumes more computational power from the basic processes.

There are limitations to this method. It is likely to provide significantly simpler solutions only to problems in need of complex consensus. Another limitation is that it is for the moment restricted to problems the solution of which can be expressed in CCS. However, recent developements show that correct backtracking mechanisms can be derived for a vastly more comprehensive SOS-based class of agent-languages [15], and that the reinterpretation theorem can be made to bear in the abstract framework of monoidal categories, and thus also covers more general grounds, such as Petri Nets [5].

The paper is self-contained but for the more technical notion of causality which is treated informally; a rigorous treatment is given in ref. [3,4]. Sec. 2 presents the self assembly specification; Sec. 3 introduces the algorithm in CCS; Sec. 4 shows that although it may deadlock, it is well designed in that its causal computations are as in the specification, and that it is therefore correct in RCCS; Sec. 5 compares with a direct solution in CCS. [4]

[4] A preliminary version of this work was presented as a poster at the 7^{th} International Conference on Artificial Evolution, Lille, France, Oct 26–28, 2005.

2 Specification

The aim of this section is to define the specification for our distributed implementation as a labelled transition system (LTS).

2.1 Transition systems and Bisimulation

A *labelled transition system* consists of a triple: a state space S, a set of labels (or actions) L, and for each $l \in L$, a binary relation over S, written $\rightarrow_l$ and called the *transition relation*. Sometimes one also adds an initial state $s_0 \in S$ to the preceding data. We will write $\rightarrow_w$, with $w = l_1 \cdots l_n$ a word over L, for the composite relation $\rightarrow_{l_1}; \cdots ; \rightarrow_{l_n}$.

Given some specification of a distributed system (such as the one given below in this section), and another LTS (possibly obtained from a CCS process as in Sec. 3) believed to be an implementation, one needs some means of stating the correctness of the implementation with respect to the specification. This is given by the notion of *bisimulation*.

Specifically, suppose given two LTSs $(S, s_0, L, \rightarrow)$, $(S', s'_0, L', \rightarrow')$, and a relation Φ over $L \times L'$. Define the *domain* of Φ as $\{l \in L \mid \exists l' \in L' : (l, l') \in \Phi\}$, and the *codomain* of Φ as the domain of the converse relation Φ^{-1}.

Given words w, w' over L, L': define w_Φ (w'_Φ) as the word w with all occurrences of labels not in the domain (codomain) of Φ erased, and write $(w, w') \in \Phi$ if $w_\Phi = l_1 \cdots l_n$ and $w'_\Phi = l'_1 \cdots l'_n$ have the same length, and for all $1 \leq i \leq n$, $(l_i, l'_i) \in \Phi$. Actions in the domain (codomain) of Φ will be called *visible*, and Φ itself will be called a *visibility relation*, thus w_Φ represents the actions in w which are visible according to Φ.

One then says a relation $\simeq$ over $S \times S'$ is a Φ-*bisimulation*, if $s_0 \simeq s'_0$, and whenever $s \simeq s'$:
– if $s \rightarrow_w t$, then $s' \rightarrow'_{w'} t'$, with $(w, w') \in \Phi$ and $t \simeq t'$,
– if $s' \rightarrow'_{w'} t'$, then $s \rightarrow_w t$, with $(w, w') \in \Phi$ and $t \simeq t'$.

The two conditions above are symmetric and state that whatever series of visible actions one LTS may perform, the other may match. In other words the two LTSs, different as they may be, are indistinguishable by synchronisation on visible actions; one says they are Φ-*bisimilar*.

In the context of CCS (see Sec. 3), one has a distinguished silent action, written τ, and setting $L = L'$, and $\Phi = \{(l, l) \mid l \neq \tau\}$ obtains what is known as weak bisimulation. Only non-silent actions, as the name suggests, are observed. An even more stringent case is when Φ is the identity relation, *i.e.*, all actions are visible, and one speaks of strong bisimulation. Our slight generalisation where the two LTSs use different sets of actions, and some flexibility is allowed regarding which actions are visible and how they match, will be convenient.

2.2 The specification

Let V be a set of nodes given together with a *degree* map $\delta : V \rightarrow \mathbb{N}$ stipulating how many nodes a given node may connect to. The trees considered here will be

represented as:

$$t ::= (a, \{t_1, \ldots, t_n\})$$

where $a \in V$ and $n \geq 0$. Hence the simplest tree is $(a, \emptyset)$ which will be simply denoted a. Other examples are $(a, \{b, c\})$ where a has two children, b and c, and $(a, \{(b, \{c\})\})$ where a and b each have one child, b and c. A childless node will be called a leaf as usual. Trees will be considered to be commutative, that is to say for instance $(a, \{b, c\})$ and $(a, \{c, b\})$ stand for the same tree, as the set notation suggests.

A tree t will be said to be *coherent* if all nodes in t have their degree as prescribed by the degree map δ, which means in particular that leaves in t will have arity smaller than 1 (and exactly 1 if they are not also the root of t). Imagine for instance that $\delta(a) = 2$, and $\delta(b) = \delta(c) = 1$, then $(a, \{b, c\})$ is coherent, while $(a, \{(b, \{c\})\})$ is not. Also one has that a is coherent if and only if $\delta(a) = 0$. Finally, we will write $n(t)$ to denote the nodes of t.

A state of our specification LTS is defined as a pair $(N, \sum_i t_i)$ where $N \subseteq V$ represent the free nodes, and each t_i is a coherent tree representing the trees already built. We write $+$ both for the addition of multisets and the disjoint union of sets. Labels are coherent trees over V, and transitions are given as follows:

$$N + n(t), \sum_i t_i \rightarrow_t N, t + \sum_i t_i$$

Note that coherence is the only constraint on trees grown out of our starting set of nodes V. Instead, one could choose a different rule for growing trees, by specifying from the outset which trees are allowed. We opt here for the local growth rule, since it allows for simpler notations, and the method given here can anyway be readily adapted to the global growth case.

3 Implementation

To define agents showing a collective behaviour in accordance with the specification given above, we use CCS [12], where the only means of communication between agents are binary synchronisations through complementary actions. This restriction translates effectively the intuitive constraint on self-assembly, namely that the global behaviour should be obtained only by means of local interaction.

3.1 CCS

CCS processes have the form:

$$p ::= 0 \mid \sum \alpha_i.p_i \mid (p \mid p) \mid (a)p \mid D(\tilde{x})$$

where $\alpha ::= a \mid \bar{a} \mid \tau$ can be a reception, an emission, or a silent action, and $D(\tilde{x})$ stands for parametric recursive definitions. Sums are taken finite, and the empty sum is denoted by 0 and called the zero process. Structural congruence, written $\equiv$, is the least equivalence relation over processes closed under sum, product and restriction, and such that sum and product are associative and commutative and

$$\text{act}\ \frac{}{\sum \alpha_i.p_i \rightarrow_{\alpha_i} p_i} \qquad \text{par}\ \frac{p \rightarrow_\alpha p'}{p \mid q \rightarrow_\alpha p' \mid q} \qquad \frac{p \rightarrow_\alpha p' \quad \alpha \neq a, \bar{a}}{(a)p \rightarrow_\alpha (a)p'}\ \text{res}$$

$$\text{syn}\ \frac{p \rightarrow_a p' \quad q \rightarrow_{\bar{a}} q'}{p \mid q \rightarrow_\tau p' \mid q'}$$

Fig. 1. CCS labelled transition system.

$$\text{NODE}_i =_{def} \tau.(\text{BUILD}_i^{\delta(i)} \mid \text{WAIT}_{i\star}^{\delta(i)}) + \sum_{j \in V} r_{ij}.(\text{BUILD}_i^{\delta(i)-1} \mid \text{WAIT}_{ij}^{\delta(i)-1}) \tag{1}$$

$$\text{BUILD}_i^{n+1} =_{def} \sum_{j \in V} \bar{r}_{ij}.\text{BUILD}_i^n,\ \text{BUILD}_i^0 =_{def} 0 \tag{2}$$

$$\text{WAIT}_{i\alpha}^{n+1} =_{def} w_i.\text{WAIT}_{i\alpha}^n,\ \text{WAIT}_{ij}^0 =_{def} \bar{w}_j.\uparrow_j^i,\ \text{WAIT}_{i\star}^0 =_{def} \underline{ok_i}.\uparrow_\star^i \tag{3}$$

Fig. 2. Self-assembly.

have 0 as neutral element. One also assumes α-conversion (renaming), and the following rule to unfold recursive definitions: $D(\tilde{x}) \equiv p$ if $D(\tilde{x}) =_{def} p$. Thereafter processes are all considered up to $\equiv$.

The CCS labelled transition system given in Fig. 1 explains how a process behaves in terms of the actions it can perform. Thus any CCS process generates an LTS, where states are processes, and labels are CCS actions.

We fix a countable subset K of CCS actions, shown as underlined in the various examples below; these are to be later interpreted as irreversible actions in RCCS, and play no specific role in the CCS semantics.

3.2 The implementation

With both our specification and agent language in place, we turn to the definition of the CCS process describing how agents interact in order to self-assemble. The definition is given in Fig. 2, with n an integer, i, $j \in V$, $\alpha \in V + \{\star\}$, and δ the degree function described earlier.

Each node is translated as a specific agent NODE_i, with $i \in V$. An agent can either decide to be the root of a new tree (left hand side of the choice in (1)), or be recruited by another agent (right hand side of the choice in (1)). In both cases, two subprocesses are spawned, BUILD_i^n, and $\text{WAIT}_{i\alpha}^n$, where n is the number of nodes the agent needs to recruit, as determined by its degree $\delta(i)$; α stands for the agent parent, if any, or for $\star$ if the agent is a root. The process BUILD_i^n (2) uses r_{ij} to recruit n free agents, while $\text{WAIT}_{i\alpha}^n$ (3) uses w_j to get confirmations of these recruitments, and then uses $\bar{w}_j$ to send a confirmation to its parent. In the special case the agent is the root of the tree, and has no parent, it performs instead the final underlined action $\underline{ok_i}$ to indicate the end of the construction.

There is no intrinsic reason why WAIT should gather confirmations in sequence; this is due to the restrictive syntax of CCS which does not allow prefixing by a set of actions (see for instance ref. [2, Sec. 3]). Likewise, using a richer language such as π-calculus [13] would make a more elegant code, replacing the r_{ij}s with a public

name (see ref. [6, Sec. 8]). That would also need a π-calculus analog of RCCS (see ref. [10, Chap. 9]), and this simple CCS version, perfectible as it is, shall be enough for our illustrative purposes.

One could set the final state of an agent to be simply a zero process, but our convention to take it to be a loop process $\uparrow^i_\alpha =_{def} \tau.\ \uparrow^i_\alpha$, indicating that agent i was successfully recruited by agent α, makes it slightly easier to extract the tree a given process has actually finished to build.

The complete system is represented as the product of all agents where all actions but the final $\underline{ok}_i$s are restricted.

3.3 Examples

Here is a computation example with $\delta(a) = 2$, $\delta(b) = \delta(c) = 1$:

$$\begin{aligned}
\textsc{node}_a \mid \textsc{node}_b \mid \textsc{node}_c \longrightarrow\ & \textsc{build}^2_a \mid \textsc{wait}^2_{a\star} \mid \textsc{node}_b \mid \textsc{node}_c \\
\longrightarrow^\star\ & \textsc{wait}^2_{a\star} \mid \textsc{wait}^0_{ba} \mid \textsc{wait}^0_{ca} \\
\equiv\ & w_a.w_a.\underline{ok_a}.\ \uparrow^a_\star \mid \bar{w}_a.\ \uparrow^b_a \mid \bar{w}_a.\ \uparrow^c_a \\
\longrightarrow^\star\ & \underline{ok_a}.\ \uparrow^a_\star \mid \uparrow^b_a \mid \uparrow^c_a \\
\longrightarrow_{\underline{ok_a}}\ & \uparrow^a_\star \mid \uparrow^b_a \mid \uparrow^c_a
\end{aligned}$$

This corresponds to a single transition $\{a, b, c\}, \emptyset \longrightarrow_{(a,\{b,c\})} \emptyset, \{(a, \{b, c\})\}$ at the specification level. In general, the construction of a tree t will decompose in $2 * n(t)$ steps. As expected, the obtained code is not correct yet, and may well deadlock, as in the following where $\delta(a) = \delta(b) = 1$, and $\delta(c) = 3$:

$$\begin{aligned}
\textsc{node}_a \mid \textsc{node}_b \mid \textsc{node}_c \longrightarrow\ & \textsc{build}^1_a \mid \textsc{wait}^1_{a\star} \mid \textsc{node}_b \mid \textsc{node}_c \\
\longrightarrow\ & \textsc{wait}^1_{a\star} \mid \textsc{node}_b \mid \textsc{build}^2_c \mid \textsc{wait}^2_{ca} \\
\longrightarrow\ & \textsc{wait}^1_{a\star} \mid \textsc{wait}^0_{bc} \mid \textsc{build}^1_c \mid \textsc{wait}^2_{ca} \\
\equiv\ & \textsc{wait}^1_{a\star} \mid \overline{w_c}.\ \uparrow^b_c \mid \textsc{build}^1_c \mid w_c.w_c.\overline{w_a}.\ \uparrow^c_a\ u \\
\longrightarrow\ & \textsc{wait}^1_{a\star} \mid \uparrow^b_c \mid \textsc{build}^1_c \mid w_c.\overline{w_a}.\ \uparrow^c_a
\end{aligned}$$

At this stage, the incoherent tree $(a, \{(c, \{b\})\})$ is built, but there is no node left for $\textsc{build}^1_c$ to recruit. Yet there is a successful trace, where a recruits b instead of c, corresponding at the specification level to the single transition $\{a, b, c\}, \emptyset \longrightarrow_{(a,\{b\})} \{c\}, \{(a, \{b\})\}$.

Therefore, it is clearly impossible to exhibit a bisimulation relation between the specification and the code induced LTS. However, the code is correct in the weaker sense that its causal computations (defined below) indeed match the specification. As we will see in the next section this is enough to ensure correctness, provided the process is re-interpreted in RCCS. The idea is that, for instance, the deadlocked trace above may backtrack in RCCS up until the wrong decision of recruiting c was made, and eventually recruit b. Note that this is not saying that the process will find

$$m ::= \langle\rangle \mid \langle i\rangle.m \mid \langle \theta, \alpha, p\rangle.m \mid \langle\!|\theta|\!\rangle.m$$
$$r, s ::= m \rhd p \mid (r \mid s) \mid (x)r$$
$$m \rhd (p \mid q) \equiv \langle 1\rangle.m \rhd p \mid \langle 2\rangle.m \rhd q$$
$$m \rhd (a)p \equiv (a)(m \rhd p) \text{ if } a \notin m$$

Fig. 3. RCCS memories, terms and additional congruence rules.

a solution, it may well loop infinitely. There are known theoretical results showing that one cannot do better in a purely non-deterministic interpretation [14]. This is of little practical importance, since such backtracking schemes will be implemented with probabilities, and such futile infinite loops will have probability zero.

To prevent backtracking from a successful state, where a coherent tree has been constructed, the corresponding underlined final actions $\underline{ok_i}$ will be chosen irreversible.

4 Correctness

This section reviews a variant of the implementation of distributed backtracking in RCCS, and the reinterpretation theorem used to derive correctness of the previous section code. The operational semantics of RCCS we present here differs from the original one [4] in that it requires abstract identifiers for uniquely tagging communications. This makes the presentation notably simpler than in the original presentation which was more adapted to the theoretical study of RCCS properties. In the present work, we adopt Phillips and Ulidowski's approach [15] in which communication keys are used to identify partners of communication. A reader familiar with the previous semantics of RCCS may wonder whether the above calculus, using communication identifiers, is equivalent to the original one. It is indeed the case [10] and we give basic hints to prove this affirmation in the appendices. Since it is not necessary for the understanding of the rest of this paper, the reader may readily proceed with the next section.

4.1 RCCS

RCCS is an extension of CCS where processes are equipped with memories used to undo computations. Memories and terms are given in Fig. 3 where: $i = 1, 2$; θ is an abstract name, drawn from a countable set $\mathcal{I}$, used to uniquely identify a communication; and p is a CCS process (as in Sec. 3) with some distinguished underlined actions declared as irreversible.

In addition to the congruence rules (see Fig. 3) for distributing memories among forking processes, and commuting restrictions with memories (assuming a was never used in the past –which is always possible using α-conversion), product and sum are considered commutative and associative, and having 0 as neutral element, as in CCS.

Define $\mathcal{I}(m)$ (resp. $\mathcal{I}(r)$) to be the set of identifiers occurring in the memory

$$\text{act}\ \frac{\theta \notin \mathcal{I}(m)}{m \rhd \alpha.p + q \rightarrow_{\theta:\alpha} \langle \theta, \alpha, q\rangle.m \rhd p} \qquad \text{act-}\ \frac{\theta \notin \mathcal{I}(m)}{\langle \theta, \alpha, q\rangle.m \rhd p \rightarrow_{\theta:\alpha-} m \rhd \alpha.p + q}$$

$$\underline{\text{act}}\ \frac{\theta \notin \mathcal{I}(m)}{m \rhd \underline{\alpha}.p + q \rightarrow_{\theta:\underline{\alpha}} \langle \theta\rangle.m \rhd p}$$

$$\text{com}\ \frac{r \rightarrow_{\theta:a} r' \quad s \rightarrow_{\theta:\bar{a}} s'}{r \mid s \rightarrow_{\theta:\tau} r' \mid s'} \qquad \text{com-}\ \frac{r \rightarrow_{\theta:a-} r' \quad s \rightarrow_{\theta:\bar{a}-} s'}{r \mid s \rightarrow_{\theta:\tau-} r' \mid s'}$$

$$\underline{\text{com}}\ \frac{r \rightarrow_{\theta:\underline{\alpha}} r' \quad s \rightarrow_{\theta:\underline{\alpha}} s'}{r \mid s \rightarrow_{\theta:\underline{\tau}} r' \mid s'}$$

$$\text{par}\ \frac{r \rightarrow_{\theta:\zeta} r' \quad \theta \notin \mathcal{I}(s)}{r \mid s \rightarrow_{\theta:\zeta} r' \mid s} \qquad \text{res}\ \frac{r \rightarrow_{\theta:\zeta} r' \quad a \notin \zeta}{(a)r \rightarrow_{\theta:\zeta} (a)r'} \qquad \text{cgr}\ \frac{r_1 \equiv r \rightarrow_{\theta:\zeta} r' \equiv r_2}{r_1 \rightarrow_{\theta:\zeta} r_2}$$

Fig. 4. RCCS labelled transition system.

m (resp. memories of subprocesses of r). The RCCS labelled transition system is given Fig. 4. Its labels are of the form $\theta : \zeta$, with $\zeta ::= \alpha \mid \alpha^- \mid \underline{\alpha}$, and θ an identifier. Side conditions of the form $\theta \notin \mathcal{I}(s)$ ensure θ is indeed unique (or a nounce in the cryptographic protocols terminology). One should note that this property corresponds to the freshness condition on communication keys in Phillips and Ulidowski's approach [15].

Forward action and communication rules each have their opposite, allowing to backtrack actions, unless the action is underlined, and thus explicitely made unbacktrackable.

4.2 Reinterpretation theorem

As said, the weaker notion of correction we need, uses the notion of causal trace. Intuitively, such traces do not involve contention among agents, since all actions therein contribute to the last one, and in addition represent atomic successful computations, since one asks the last action to be the trace only irreversible one.

More precisely, a trace σ is said to be *causal* if it contains a single irreversible transition $\underline{t}$ and for all $\sigma' \sim \sigma$, σ' ends by $\underline{t}$, where $\sim$ is the equivalence relation over CCS traces obtained by permuting concurrent transitions [1].
Here are some examples:

$$a.\underline{b}.0 \mid c.0 \ \rightarrow_a\ \underline{b}.0 \mid c.0 \ \rightarrow_c\ \underline{b}.0 \ \rightarrow_{\underline{b}}\ 0$$
$$a.\underline{b}.0 \mid c.0 \ \rightarrow_a\ \underline{b}.0 \mid c.0 \ \rightarrow_{\underline{b}}\ c.0$$
$$a.\underline{b}.0 \mid \bar{a}.0 \ \rightarrow_\tau\ \underline{b}.0 \ \rightarrow_{\underline{b}}\ 0$$

The first trace is not causal since its last action $\underline{b}$ commutes to the earlier action c, as in the second one which is causal; likewise, the last trace is causal, since the

marked action $\underline{b}$ does not commute to τ.

Definition 4.1 Let P be the set of CCS processes, K be the set of underlined CCS actions, and define $p_1 \rightarrow^c_{\underline{k}} p_2$, if there is a causal trace from p_1 to p_2 ending with $\underline{k}$. The *causal transition system* induced by p, written $CTS(p)$, is defined as $(P, p, K, \rightarrow^c_k)$.

In the examples above, one has $a.\underline{b}.0 \mid c.0 \rightarrow^c_{\underline{b}} c.0$, $a.\underline{b}.0 \mid \bar{a}.0 \rightarrow^c_{\underline{b}} 0$, and *not* $a.\underline{b}.0 \mid c.0 \rightarrow^c_{\underline{b}} 0$.

The theorem below asserts that the LTS induced by the interpretation of p in RCCS is equivalent to $CTS(p)$, when observations are restricted to irreversible actions.

Theorem 4.2 ([4]) *Let p be a CCS process, and Φ be the relation $\{(\underline{k}, \theta : \underline{k}); \underline{k} \in K, \theta \in \mathcal{I}\}$, then $CTS(p) \approx_\Phi LTS(\langle\rangle \rhd p)$.*

4.3 Back to self assembling trees

To apply this definition to the case of interest, we need to map macro-states (states of the specification) to micro-states (states of the corresponding process). Define first the family of maps $[\![\_]\!]_\alpha$, with $\alpha \in V + \{\star\}$:

$$[\![(a, \{t_1, \ldots, t_n\})]\!]_\alpha =_{def} \uparrow^a_\alpha \mid [\![t_1]\!]_a \mid \ldots \mid [\![t_n]\!]_a$$

This obtains a map from macro-states to what one might call their standard representation as micro-states (restrictions are not shown):

$$[\![N, \textstyle\sum_i t_i]\!] =_{def} \prod_{i \in N} \textsc{node}_i \mid \prod_j [\![t_j]\!]_\star$$

Defining $\Phi' = \{(t, \underline{ok_i}) \mid i \in V\}$, one has:

Proposition 4.3 *The relation $\{(N, \sum_i t_i), [\![N, \sum_i t_i]\!]\}$ is a Φ'-bisimulation between the specification LTS and $CTS([\![V]\!])$.*

The proof is routine. Concretely, this is saying two things. Firstly, whenever some tree may be constructed from the remaining free nodes of the specification, there exists a causal sequence of interactions among the agents that implements it (see first example in Sec. 3). Secondly, whenever a tree is built after a successful series of agent interactions, this tree is indeed coherent, and therefore corresponds to a transition in the specification (this is even easier to prove, since the number of neighbours of any given process representing a node is always kept smaller or equal to its arity as specified by δ).

Putting that proposition together with the theorem above one obtains:

Corollary 4.4 *The specification LTS and $LTS(\langle\rangle \rhd [\![V]\!])$, are $\Phi'; \Phi$-bisimilar.*

One may object that the visibility relation $\Phi'; \Phi$ used here is highly non-injective, since it relates a tree t to some ok_i, which contains no other information than the name of the process being the root of t. Using a value-passing version of CCS, one can decorate the implementation and construct during the assembly an expression

$$
\begin{aligned}
\textsc{node}_i &=_{def} \tau.\textsc{build}_{i\star}^{\delta(i),\emptyset} + \sum_{j\in I} r_{ij}.\textsc{build}_{ij}^{\delta(i),\emptyset} \\
\textsc{build}_{ij}^{n+1,S} &=_{def} \sum_{k\in I} \bar{r}_{ik}.\textsc{build}_{ij}^{n,S\cup\{k\}} + kill_i.\textsc{abort}_i^S \\
\textsc{build}_{i\star}^{n+1,S} &=_{def} \sum_{k\in I} \bar{r}_{ik}.\textsc{build}_{ij}^{n,S\cup\{k\}} + \tau.\textsc{abort}_i^S \\
\textsc{build}_{i\alpha}^{0,S} &=_{def} \textsc{wait}_{i\alpha}^{|S|,S} \\
\textsc{wait}_{ij}^{n+1,S} &=_{def} w_i.\textsc{wait}_{ij}^{n,S} + \ kill_i.\textsc{abort}_i^S \\
\textsc{wait}_{i\star}^{n+1,S} &=_{def} w_i.\textsc{wait}_{i\star}^{n,S} + \tau.\textsc{abort}_i^S \\
\textsc{wait}_{ij}^{0,S} &=_{def} \overline{w}_j.\uparrow_{ij}^S + kill_i.\textsc{abort}_i^S \\
\textsc{wait}_{i\star}^{0,S} &=_{def} \underline{ok_i}.\uparrow_{i\star}^S \\
\textsc{free}^{S\cup\{i\}}(end) &=_{def} \overline{kill}_i.\textsc{free}^S(end) \\
\textsc{free}^{\emptyset}(end) &=_{def} \overline{end}.0 \\
\uparrow_{ij}^S &=_{def} \tau.\uparrow_{ij}^S + kill_i.\textsc{abort}_{ij}^S \\
\uparrow_{i\star}^S &=_{def} \tau.\uparrow_{i\star}^S \\
\textsc{abort}_i^S &=_{def} (end)(\textsc{free}^S(end) \mid end.\textsc{n}_i)
\end{aligned}
$$

Fig. 5. Self-assembly directly in CCS.

describing the tree being constructed, which could then be used to encode injectively t in the final irreversible action concluding the construction. However, the bisimulation relation we exhibit clearly contains all the needed information since the macro-to-micro map itself is injective.

5 Discussion

It remains to appreciate whether a direct solution in CCS could compare well with the indirect solution we have obtained. We base our discussion on a comparison with one particular reasonable direct implementation, given Fig. 5, and obtained by patching the indirect code to recover from deadlocks. The recruitment phase is quite similar to the one in the previous code, except BUILD and WAIT processes are now run in sequence. A more important difference is that the root may abort the construction by running at any time the process $\textsc{abort}_i^S$ which waits for the $\textsc{free}^S(end)$ process to free recruited agents, and then re-spawns the initial state. Any already recruited agent i enters the abort state upon reception of a request by its parent using action $kill_i$. Accordingly, the final state $\uparrow_{i\alpha}^S$ indicating that the i^{th} agent has finished its part of the recruitment, in the case $\alpha \neq \star$ still waits for a possible such abort request initiated by the root agent and forwarded by its parent.

Thus, the direct code may escape deadlocks. To keep things simple, we give up part of the distributed structure of the system: a node does not wait for the confirmations of its children until it has completed its recruiting task. This results

in a better control of the construction process at the price of a loss of efficiency, since no agent can validate its recruitment until its parent is ready to receive the validation. Yet the main difference is in the backtracking mechanism: the RCCS code finds its way to a final shape by using partial backtracking, whereas the CCS one uses a top-down cancellation procedure to abort altogether the construction (as in ref. [9]).

One sees the RCCS code is more intuitive; this is because, in essence, it is easier to describe what has to be done, than what has to be undone. Furthermore, it is necessary to prove that the complete code conforms to its specification, and exhibit a bisimulation relation between the code and the specification (given Sec. 2). It is not clear at all how to do this by hand, and to get a sense of how difficult that may be, we have tested our code with the Mobility Workbench [16], a toolkit able to verify certain properties on π-calculus [13] processes. We succeeded in building the bisimulation relation for a system composed of 3 agents. For such a simple system, the Mobility Workbench already returns 600 states. Running the tool for 24 hours was not enough to obtain an answer in the case of a system of 4 agents. [5] The reason for this explosion in the size of the bisimulation is that the backtracking mechanism induces a lot of transitory states that try to undo their local constructions. More details about how the indirect method helps in automated verification can be found in ref. [11].

6 Conclusion and future work

We have presented a distributed algorithm for self assembling trees using CCS. Part of the appeal of the solution is that both the language used and the solution itself stay simple. First one formulates a solution which is only required to be correct in weak sense. One then uses the reversible infrastructure provided by RCCS to obtain correctness. Not only the proof is greatly simplified in so doing, but the actual code obtained is also simpler in that backtracking stays implicit.

Our model leaves aside more subtle forms of self-assembly based on graph-rewriting. These would likely need a more powerful language [7,6], but there seems to be no reason why the decomposition of the self-assembly question advocated in this paper, would not extend to these richer languages. Our model also ignores the question of how one represents real space, in that connections are represented abstractly as synchronisations. Another important aspect of self-assembly which our model does not take into account is its quantitative nature, as our model only knows of non-deterministic evolutions, and doesn't assign to them any measure of their likelihood. More work is needed to understand how both spatial and probabilistic features could be added to the picture. One could think of a distributed language where agents would use timeouts to decide to backtrack. Substituting the RCCS operational semantics to the ordinary CCS one, or whichever richer language one is using, would obtain agents that would behave correctly with respect to the global specification. This requires first a thorough study of the impact of timeouts on the operational semantics of RCCS, a question which we plan to address in future work.

[5] Tests were made with a 1.4 GHz Pentium M with 256 MB of RAM.

Decoupling in a given system the forward and backward components of its behaviour, is even more natural in the modelling and analysis of biomolecular interactions. Indeed, one may regard molecules as blind agents trying to bind haphazardly. Each time their spatial configurations match, proteins have a chance to bind, and these bounds are also frequently broken down. These exploration mechanisms have been argued to be of central importance in the evolvability of biological systems [8]. Here the implicit backtracking mechanism of RCCS comes in handy as a transparent way to model this instability [2], but, if anything, the addition of probabilities to backward moves, so as to generate a quantitative behaviour and be able to tune the backtracking mechanism, seems even more important in this specific context, and it remains to be seen how the method we have illustrated here can cope with these.

References

[1] Boudol, G. and I. Castellani, *Permutation of transitions: An event structure semantics for CCS and SCCS*, in: de Bakker, J.W. et al., editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, LNCS **354** (1989), pp. 411–427.

[2] Danos, V. and J. Krivine, *Formal molecular biology done in CCS*, in: *Proceedings of BIO-CONCUR'03, Marseilles, France*, ENTCS (2003), to appear.

[3] Danos, V. and J. Krivine, *Reversible communicating systems*, in: *CONCUR'04*, LNCS **3170** (2004), pp. 292–307.

[4] Danos, V. and J. Krivine, *Transactions in RCCS*, in: *CONCUR'05*, LNCS **3653** (2005).

[5] Danos, V., J. Krivine and P. Sobocinski, *General reversibility*, in: *EXPRESS'06*, ENTCS (2006), to appear.

[6] Danos, V. and C. Laneve, *Formal molecular biology.*, TCS **325** (2004), pp. 69–110.

[7] Danos, V. and F. Tarissan, *Self-assembling graphs*, in: J. Mira and J. Alvarez, editors, *IWINAC'05*, LNCS **3561** (2005), pp. 501–510.

[8] Kirschner, M. and J. Gerhart, *Evolvability*, PNAS **95** (1998), pp. 8420–8427.

[9] Klavins, E., *Automatic synthesis of controllers for assembly and formation forming*, in: *ICRA'02*, 2002, pp. 3296–3302.

[10] Krivine, J., "Algèbres de Processus Réversibles," Ph.D. thesis, Université Paris 6 & INRIA-Rocquencourt (2006), moscova.inria.fr/~krivine.

[11] Krivine, J., *A verification algorithm for declarative concurrent programming*, Technical report, INRIA-Rocquencourt (2006), moscova.inria.fr/~krivine.

[12] Milner, R., "Communication and Concurrency," International Series on Computer Science, Prentice Hall, 1989.

[13] Milner, R., J. Parrow and D. Walker, *A calculus of mobile process (*i *and* ii*)*, Information and Computation **100** (1992), pp. 1–77.

[14] Palamidessi, C., *Comparing the expressive power of the synchronous and asynchronous pi-calculi*, MSCS **13** (2003), pp. 685–719.

[15] Phillips, I. and I. Ulidowski, *Reversing algebraic process calculi*, in: *FOSSACS'06*, LNCS **3921** (2006), pp. 246–260.

[16] Victor, B. and F. Moller, *The Mobility Workbench — a tool for the π-calculus*, in: D. Dill, editor, *CAV'94*, LNCS **818** (1994), pp. 428–440.

7 Appendix

Instead of abstract names, one can use memories as concrete identifiers [3]. We recall in this appendix how this is done, and argue that both the abstract and concrete identifying schemes are in fact intertranslatable. This is useful in so far as the reinterpretation theorem we used earlier was actually proven only for the concrete scheme. A complete proof is in ref. [10, Chap. 3].

Concrete memories are given as:

$$m ::= \langle\rangle \mid \langle i\rangle \cdot m \mid \langle \star, \alpha, p\rangle \cdot m \mid \langle m', \alpha, p\rangle \cdot m \mid \langle\!|\circ|\!\rangle \cdot m$$

where $\star$ stands for an unknown communication partner, the equivalent of which, in the semantics above, is a θ that is unique to the whole process. The corresponding transition system, shown below, has now labels of the form $\mu : \zeta$ where μ is a set of one or two memories; $r_{m'@m}$ denotes the substitution of $\star$ with the concrete identifier m' in $\langle \star, \alpha, p\rangle \cdot m$; irreversible rules are not shown.

$$\frac{}{m \rhd \alpha.p + q \to_{m:\alpha} \langle \star, \alpha, q\rangle \cdot m \rhd p} \qquad \frac{}{\langle \star, \alpha, q\rangle \cdot m \rhd p \to_{m:\alpha^-} m \rhd \alpha.p + q}$$

$$\frac{r \to_{m:\bar{a}} r' \quad s \to_{m':a} s'}{r \mid s \to_{m,m':\tau} r'_{m'@m} \mid s'_{m@m'}} \qquad \frac{r \to_{m:\bar{a}^-} r' \quad s \to_{m':a^-} s'}{r_{m'@m} \mid s_{m@m'} \to_{m,m':\tau^-} r' \mid s'}$$

$$\frac{r \to_{\mu:\zeta} r'}{r \mid s \to_{\mu:\zeta} r' \mid s} \qquad \frac{r \to_{\mu:\zeta} r' \quad \zeta \neq a, \bar{a}, a^-, \bar{a}^-}{(a)r \to_{\mu:\zeta} (a)r'} \qquad \frac{r \equiv r_1 \to_{\mu:\zeta} r_2 \equiv r'}{r \to_{\mu:\zeta} r'}$$

Given an abstract process r, and assuming any identifier occurs at most twice in r, the following defines inductively a map M_r from an abstract process to a concrete one (all other clauses being trivial):

$$M_r(\langle\!|\theta|\!\rangle \cdot m) = \langle\!|\circ|\!\rangle \cdot M_r(m)$$

$$M_r(\langle\theta, \alpha, p\rangle \cdot m) = \begin{cases} \langle M_r(m'), \alpha, p\rangle \cdot M_r(m) & \text{if } \langle\theta, \bar{\alpha}, q\rangle \cdot m' \in r \\ \langle \star, \alpha, p\rangle \cdot M_r(m) & \text{else} \end{cases}$$

Conversely, given a μ indexed family of identifiers θ_μ such that $\theta_\mu \neq \theta_{\mu'}$ if $\mu \cap \mu' \neq \mu$, one can map concrete processes to abstract ones (again all other clauses are trivial):

$$\Theta(\langle\!|\circ|\!\rangle \cdot m) = \langle\!|\theta_{\{m\}}|\!\rangle \cdot \Theta(m)$$

$$\Theta(\langle m, \alpha, p\rangle \cdot m') = \langle\theta_{\{m,m'\}}, \alpha, p\rangle \cdot \Theta(m')$$

$$\Theta(\langle \star, \alpha, p\rangle \cdot m) = \langle\theta_{\{m\}}, \alpha, p\rangle \cdot \Theta(m)$$

We suppose now all concrete processes have unique memories, and all abstract processes have identifiers occurring at most twice. This is easily shown to be preserved under computations.

Proposition 7.1 *If $r \to_{\theta:\zeta} s$ then $\exists \mu : M_r(r) \to_{\mu:\zeta} M_s(s)$ and if $r \to_{\mu:\zeta} s$ then $\exists \theta : \Theta(r) \to_{\theta:\zeta} \Theta(s)$.*

For the first implication: if $r \rightarrow_{\theta:\tau} s$, take $\mu = \{M_s(m_1), M_s(m_2)\}$ where $\langle \theta, a, p\rangle \cdot m_1$, $\langle \theta, \bar{a}, q\rangle \cdot m_2 \in s$; if $r \rightarrow_{\theta:\tau-} s$, take $\mu = \{M_r(m_1), M_r(m_2)\}$ where $\langle \theta, a, p\rangle \cdot m_1$, $\langle \theta, \bar{a}, q\rangle \cdot m_2 \in r$. For the second implication, it suffices to take $\theta = \theta_\mu$. The side condition in the par rule (see Fig. 4) holds thanks to the unicity of memories and the assumption that $\theta_\mu \neq \theta_{\mu'}$ whenever $\mu \cap \mu' \neq \mu$.