The Replacement Operation for CCP Programs

Marco Bertolino[1], Sandro Etalle[2], and Catuscia Palamidessi[3]

[1] S.I.A.C. bertolino@siac.it
[2] Universiteit Maastricht ealle@cs.unimaas.nl
[3] Penn State University catuscia@cse.psu.edu

Abstract. The *replacement* is a very powerful transformation operation which – both within the functional paradigm as well as within the logic programming one – can mimic the most common transformation operations such as unfold, fold, switching, distribution. Because of this flexibility, it can be incorrect if used without specific applicability conditions.
In this paper we present applicability conditions which ensure the correctness of the replacement operation in the context of Concurrent Constraint Programs. Furthermore we show that, under these conditions, the replacement generalizes both the unfolding operation as well as a restricted form of folding operation.

1 Introduction

Concurrent constraint programming ([26]) (ccp, for short) is a concurrent programming paradigm which derives from replacing the *store-as-valuation* concept of von Neumann computing by the *store-as-constraint* model. The computational model of ccp is based on a global *store*, represented by a constraint, which expresses some partial information on the values of the variables involved in the computation. The concurrent execution of different processes, which interact through the common store, refines the partial information of the values of the variables by adding (*telling*) constraints to the store. Communication and synchronization are achieved by allowing processes to test (*ask*) if the store entails a constraint before proceeding in the computation.

Central to the development of large and efficient applications is the study of optimization techniques. To this end, while there exists a history and a wide literature on transformations for sequential languages, ranging from theoretical studies to implemented tools, there are only few and relatively recent attempts to apply these techniques to concurrent languages. To the best of our knowledge, the only papers addressing this issue are [10, 11, 29, 23, 14, 17, 13, 9]. In our opinion, this situation can be ascribed to the non-determinism and the synchronization mechanisms present in concurrent languages, which substantially complicate their semantics. In this context, transformation techniques have to employ more sophisticated analysis tools.

The area closest to ccp with a large literature on transformation operations is the area of Constraint Logic Programs (CLP). For this paradigm, the literature on transformations can be divided into two main branches. On one hand we find methods which focus exclusively on the manipulation of the constraint for compile-time [18, 19] and for low-level local optimization [15]. On the other hand there are techniques such as the unfold/fold transformation systems, which were developed initially for Logic Programs [28] and then applied to CLP [16, 1, 8] and to ccp in [9]. These ones focus primarily on the declarative side of the program.

The *Replacement* is a program transformation technique flexible enough to encompass both the above kinds of optimization: it can be profitably used to manipulate both the constraint and the "declarative" side of a program. In fact the replacement operation, which was introduced in the field of Logic Programming by Tamaki and Sato [28] and later further developed and applied to CLP in [16, 1, 7], syntactically consists in replacing an agent in the body of a program definition by another one. It is therefore a very general operation and it is able to mimic many other transformations, such as thinning, fattening [3] and folding. In the logic programming area, a lot of research [4, 5, 1, 6, 7, 12, 16, 22, 28, 27] has been devoted to the definition of *applicability conditions* sufficient to guarantee the correctness of replacement w.r.t. several different semantics. See [21] for a survey on transformation techniques for logic languages.

The goal of this paper is to provide some natural and relatively simple applicability conditions which ensure the correctness of the replacement for ccp, i.e. that the transformed program is equivalent to the original one. Of course, the notion of equivalence depends on the semantics one refers to. In the case of ccp programs the notion of semantics usually considered is the set of final stores, i.e. the stores that can be obtained at the end of a computation (we say that a computation ends when it cannot proceed anymore). These are often called "observables". Sometimes also the stores obtained as limit of the intermediate stores in an infinite computation are considered part of the observables, but in this paper we will not take them into account. In this paper we will actually consider a stronger semantics, namely *simulation equivalence*, or *reciprocal simulation*. The reason for this choice is that observables-equivalence, in a concurrent context, is too weak for ensuring the correctness of transformation techniques such as the replacement. Basically this is due to lack of compositionality. When replacing an agent A with A' we will require that the two agents be *simulation-equivalent*. Ultimately, what we aim at is a set of conditions which guarantee that the program resulting from the replacement operation is simulation-equivalent to the original one, in which case we say that the operation is *correct*. Since simulation-equivalence implies observables-equivalence, correctness implies that the two programs are also observables-equivalent.

It turns out that the simulation-equivalence of the replacing and the replaced agents alone is not sufficient to guarantee the correctness of the operation. In fact, we will show that it guarantees only *partial* correctness i.e. that the resulting program is simulated by the original one, but not vice-versa. In order to guarantee total correctness we have to ensure that *the replacement must not introduce unwanted loops*. Now, consider for example the contrived program:

$$\begin{array}{lll} p(x) & :- & q(x) \\ q(x) & :- & (\ \ ask(x = [\ ]) \rightarrow \mathbf{stop} \\ & & \ \ + ask(\exists_{y,ys}\ x = [y|ys]) \rightarrow \exists_{y,ys}\ tell(x = [y|ys]) \parallel p(ys)) \end{array}$$

Here, $q(x)$ is clearly equivalent to $p(x)$; but if we replace $q(x)$ with $p(x)$ in the body of the first definition we obtain the definition $p(x)$:- $p(x)$, which is certainly not equivalent to the original one.

To avoid unwanted loops, in this paper we follow the inspiration of [4, 7, 25] (which focus on logic programs, CLP and functional programs, respectively), and we individuate two situations in which the operation certainly does not introduce any unwanted loop: The simplest one is (a): *when the replacing agent is independent from the definition that is going to be transformed*. This is the case in which the definition of the replacing agent does not depend on the definition being transformed.

Clearly, this condition is sufficient to guarantee that no extra loops are introduced by the transformation. For instance, it rules out the situation described above, in which we replaced $q(x)$ with $p(x)$. Moreover, it is immediate to check. The disadvantage is that it clearly does not allow to introduce recursion inside a definition: For instance in the above example we might want to replace $p(ys)$ with $q(ys)$. Notice that this replacement *does* introduce a loop in the form of a direct recursion inside the definition of p. Such a step would be clearly forbidden by the condition (a) above. In order to be able to perform also this second replacement we need an applicability condition alternative to (a). Here we provide such an alternative condition, namely, (b): *when the replacing agent is at least as efficient as the replaced one*. Referring to the simulation semantics this is the case when the following holds: if the replaced agent can compute an "answer" constraint c with a transition sequence which uses n procedure expansions, then the replacing agent can also compute the answer c for the replacing one in m procedure expansions with $m \leq n$. This is undoubtedly a desirable situation which fits well in the natural context in which the transformation is performed in order to increase program's execution speed. Moreover, in the above example we have that $q(ys)$ is equivalent to $p(ys)$ and more efficient than $p(ys)$, because it requires one procedure expansion less to "reach" the same answer. Thus condition (b) is flexible enough to allow the above replacement and therefore to introduce recursion, which can be seen as an example of *wanted* loop.

2 Preliminaries

In this section we briefly recall the definition of ccp. We refer to [26] for more details. The language ccp is parametric w.r.t. a cylindric constraint system $\mathbf{C} = \langle \mathcal{C}, \leq, \wedge, true, false, Var, \exists, \delta \rangle$. Roughly speaking, such system represents a set of logical formulas $\mathcal{C}$ (constraints) closed under conjunction ($\wedge$) and existential quantifier ($\exists$). The relation $\leq \subseteq \mathcal{C} \times \mathcal{C}$ is an ordering relation whose inverse represents the notion of *logical entailment*. *Var*, with typical elements $x, y, \ldots$, is the set of variables which can appear in the constraints. δ is a function from $Var \times Var$ into constraints, which gives *true* on all the pairs of identical variables (diagonal elements). Intuitively, δ_{xy} represents the equality constraint between x and y. In the following, the notation $\tilde{\chi}$ indicates a sequence of the form $\chi_1, \ldots, \chi_n$. The processes are described by the following grammar

$$\begin{aligned} \textit{Processes } P &::= [D, A] \\ \textit{Declarations } D &::= \epsilon \mid p(\tilde{x}) \text{ :- } A \mid D, D \\ \textit{Agents } A &::= \mathbf{stop} \mid \mathbf{tell}(c) \mid \textstyle\sum_{i=1}^{n} \mathbf{ask}(c_i) \rightarrow A_i \mid A \parallel A \mid \exists_x A \mid p(\tilde{x}) \end{aligned}$$

The agent **stop** represents successful termination. The basic actions are given by $\mathbf{ask}(c)$ and $\mathbf{tell}(c)$ constructs, where c is a *finite constraint*, i.e. an algebraic element of $\mathcal{C}$. These actions work on a common *store* which ranges over $\mathcal{C}$. $\mathbf{ask}(c)$ is a test on the current store and its execution does not modify the store. We say that $\mathbf{ask}(c)$ is a *guard* and that is *enabled* in d iff $c \leq d$. If d is the current store, then the execution of $\mathbf{tell}(c)$ sets the store to $c \wedge d$. The *guarded choice* agent $\sum_{i=1}^{n} g_i \rightarrow A_i$ selects nondeterministically one g_i which is enabled, and then behaves like A_i. If no guards are enabled, then it *suspends*, waiting for other (parallel) agents to add information to the store. Parallel composition is represented by $\parallel$. The situation in which all components of a system of parallel agents suspend is called *global suspension* or *deadlock*. The agent $\exists_x A$ behaves like A, with x considered *local* to A. Finally, the agent $p(\tilde{x})$ is a procedure call, where p is the name of the procedure and $\tilde{x}$ is the list of the actual parameters. The meaning of $p(\tilde{x})$ is given by a procedure declaration of the form $p(\tilde{y})$:- A, where $\tilde{y}$ is the list of the formal parameters. A set of declarations constitutes a *program*.

The operational model of ccp, informally introduced above, is described by a transition system $T = (\textit{Conf}, \longrightarrow)$, where the set of configurations is defined as $\textit{Conf} = \textit{Processes} \times \mathcal{C}$. Sometimes we will need to indicate the the number (0 or 1) of procedure expansions taking place during a transition. In that case we will use the notation $\longrightarrow_n$, where n is the number of procedure expansions. Table 1 describes the rules of T.

The guarded choice operator models global non-determinism (**R2**), in the sense that it depends on the current store whether or not a guard is enabled, and the current store is subject to modifications by the external environment (**R1**). **R3** and **R4** describe parallelism as interleaving. To describe locality (**R5**) the syntax has been extended by an agent $\exists_x^d A$ in which x is local to A and d is the store that has been produced locally on x. Initially the local store is empty, i.e. $\exists_x A = \exists_x^{true} A$. The procedure expansion is modeled by **R6**. $\Delta_{\tilde{y}}^{\tilde{x}}$ stands for $\exists_{\tilde{\alpha}}^{\delta_{\tilde{x}\tilde{\alpha}}} \exists_{\tilde{y}}^{\delta_{\tilde{\alpha}\tilde{y}}}$ and it is used to establish the link between the formal parameters $\tilde{y}$ and the actual parameters $\tilde{x}$. The notation $\delta_{x_1,\ldots,x_n\alpha_1,\ldots,\alpha_n}$ represents the conjunction of the constraints $\delta_{x_1\alpha_1}, \ldots, \delta_{x_n\alpha_n}$. The variables $\tilde{\alpha}$ are introduced in order to avoid problems related to name clashes between $\tilde{x}$ and $\tilde{y}$; they are assumed to occur neither in the procedure declaration nor in the procedure call.

Note that in a transition, only the agent part of the process is modified. Namely, if $\langle [D, A], c \rangle \longrightarrow \langle [D', A'], c' \rangle$, then $D' = D$. However, in order to define the notion of simulation, we find it convenient to have the program explicit in the configuration.

R1	$\langle [D, \mathbf{tell}(c)], d\rangle \longrightarrow_0 \langle [D, \mathbf{stop}], c \wedge d\rangle$	
R2	$\langle [D, \sum_{i=1}^{n} g_i \rightarrow A_i], d\rangle \longrightarrow_0 \langle [D, A_j], d\rangle$	$j \in [1, n]$ *and* $g_j = \mathbf{ask}(c)$ *and* $c \leq d$
R3	$\dfrac{\langle [D, A], c\rangle \longrightarrow_n \langle [D, A'], c'\rangle}{\langle [D, B \parallel A], c\rangle \longrightarrow_n \langle [D, B \parallel A'], c'\rangle}$	
R4	$\dfrac{\langle [D, A], c\rangle \longrightarrow_n \langle [D, A'], c'\rangle}{\langle [D, A \parallel B], c\rangle \longrightarrow_n \langle [D, A' \parallel B], c'\rangle}$	
R5	$\dfrac{\langle [D, A], d \wedge \exists_x c\rangle \longrightarrow_n \langle [D, B], d'\rangle}{\langle [D, \exists_x^d A], c\rangle \longrightarrow_n \langle [D, \exists_x^{d'} B], c \wedge \exists_x d'\rangle}$	
R6	$\langle [D, p(\tilde{x})], c\rangle \longrightarrow_1 \langle [D, \Delta_{\tilde{y}}^{\tilde{x}} A], c\rangle$	$p(\tilde{y})$:- $A \in D$

Table 1. The transition system T.

We describe now what we intend to *observe* about a process. Intuitively, for every possible initial store (input) we want to collect the results (outputs) of all possible finite computations.

Definition 1 (Observables). *Given a process P, we define its* observables *as follows:*

$$\mathcal{O}(P) = \{\langle c, c'\rangle \mid \text{ there exists } P' \text{ s.t. } \langle P, c\rangle \longrightarrow^* \langle P, c'\rangle \not\longrightarrow\}$$

where $\not\longrightarrow$ denotes the absence of outgoing transitions and $\longrightarrow^*$ denotes the reflexive and transitive closure of $\longrightarrow$. □

In some cases we will need to refer to the number of procedure expansions taking place during a sequence of transitions. We will then use the notation $\longrightarrow_n^*$, where $n \geq 0$ represents such number. Formally:

$$\langle P, c\rangle \longrightarrow_0^* \langle P, c\rangle$$
$$\langle P, c\rangle \longrightarrow_{n+m}^* \langle P', c'\rangle \text{ if } \langle P, c\rangle \longrightarrow_n \langle P'', c''\rangle \text{ and } \langle P'', c''\rangle \longrightarrow_m^* \langle P', c'\rangle$$

3 Simulation

In this section we introduce the notion of simulation for ccp programs and agents. This will be the key semantic concept throughout the paper, and will allow us to characterize – among other things – whether a transformation is correct or not.

The main reason why we introduce the notion of simulation is that it has strong properties (like compositionality) that are crucial for proving the correctness of the replacement. Moreover, simulation semantics is correct with respect to the observables.

Our notion of simulation is inspired by the homonymous notion in theory of concurrency, see for instance [30]. In process algebras like CCS simulation equivalence is considered too weak, because it is not correct w.r.t. maximal trace semantics, which is the standard notion of observables. Researchers in concurrency theory usually consider, instead, the stricter notion of *bisimulation* [20]. In the case of ccp, however, the standard notion of observable (final stores) is more abstract, and the asynchronous nature of communication allows us to maintain a more abstract equivalence also when considering the issue of compositionality, in the sense that less information is needed to achieve the closure under contexts. Furthermore, being a weaker relation, simulation has the advantage of allowing us to use transformation rules which are applicable in more cases.

In our definition of simulation for ccp, the main differences w.r.t. the classical notion is that we consider the simulation at the level of the stores rather than of the actions. This is because of the asynchronous nature of ccp communication: the relevant changes during a computations are those made in the store; actions are relevant only for their effect on the store. This is reflected by the definition of the transition system: the transition relation is unlabeled and the configurations contain the store. In CCS, on the contrary, transitions are labeled by actions and there is no concept of store. Another difference is that we impose a condition on terminal configurations: when a process terminates we require that the simulating process eventually terminates as well and produces the same store. We introduce this condition, which has no analogous in the classical notion of simulation, in order to achieve correctness w.r.t. the observables. Finally, we require a simulation to be joint-closed (see below). Again, this condition has no counterpart in CCS, and it is needed here to ensure compositionality.

Definition 2. *A relation* $\mathcal{R} \subseteq \mathrm{Conf} \times \mathrm{Conf}$ *is* joint-closed *iff for every* $d \in \mathcal{C}$ *and every pair* $(\langle Proc, c\rangle, \langle Proc', c'\rangle) \in \mathcal{R}$ *we have that* $(\langle Proc, c \wedge d\rangle, \langle Proc', c' \wedge d\rangle) \in \mathcal{R}$. $\square$

Recall that a *program* is a set of declarations. In the sequel we indicate programs by D, D' etc., and processes by P, P' etc.

Definition 3 (Simulation on agents). *A relation* $\mathcal{S} \subseteq \mathrm{Conf} \times \mathrm{Conf}$ *is a* simulation *iff it is joint-closed, and for every pair* $(\langle [D_1, A_1], c_1\rangle, \langle [D_2, A_2], c_2\rangle)$ *in* $\mathcal{S}$ *the following two conditions hold:*

(i) If $\langle [D_1, A_1], c_1\rangle \longrightarrow \langle [D_1, A_1'], c_1'\rangle$, *then for some* A_2', c_2', $\langle [D_2, A_2], c_2\rangle \longrightarrow^* \langle [D_2, A_2'], c_2'\rangle$, $c_2' \geq c_1'$, *and* $(\langle [D_1, A_1'], c_1'\rangle, \langle [D_2, A_2'], c_2'\rangle) \in \mathcal{S}$.

(ii) If $\langle [D_1, A_1], c_1\rangle \not\longrightarrow$, *then for some* A_2', $\langle [D_2, A_2], c_2\rangle \longrightarrow^* \langle [D_2, A_2'], c_1\rangle \not\longrightarrow$. $\square$

As a notational convention, if $(\langle P, c\rangle, \langle P', c'\rangle) \in \mathcal{S}$ for some simulation $\mathcal{S}$, we say that $\langle P', c'\rangle$ *simulates* $\langle P, c\rangle$ and we write $\langle P, c\rangle \trianglelefteq \langle P', c'\rangle$. Furthermore, we say that the process P' *simulates* P, notation $P \trianglelefteq P'$, iff $\langle P, true\rangle \trianglelefteq \langle P', true\rangle$. Note that, by joint-closedness, $P \trianglelefteq P'$ implies that $\langle P, c\rangle \trianglelefteq \langle P', c\rangle$ for every $c \in \mathcal{C}$.

The following proposition states the correctness of simulation w.r.t. the observables:

Proposition 4. *If $P \trianglelefteq P'$, then $\mathcal{O}(P) \subseteq \mathcal{O}(P')$.*

Proof (Sketch) Let $\langle c, d\rangle \in \mathcal{O}(P)$. Then there exist $P_0, P_1, \dots P_n$ and $c_0, c_1, \dots c_n$, with $n \geq 0$, such that

$$\langle P, c\rangle = \langle P_0, c_0\rangle \longrightarrow \langle P_1, c_1\rangle \longrightarrow \dots \longrightarrow \langle P_n, c_n\rangle = \langle P_n, d\rangle \not\longrightarrow$$

Since $P \trianglelefteq P'$, one can show by induction on n that there exist $P'_0, P'_1, \dots P'_n, P'_{n+1}$ and $c'_0, c'_1, \dots c'_n$ such that $\forall i \in \{1, \dots, n\}$, $\langle P_i, c_i\rangle \trianglelefteq \langle P'_i, c'_i\rangle$, $c_i \leq c'_i$, and

$$\langle P', c\rangle = \langle P'_0, c'_0\rangle \longrightarrow^* \langle P'_1, c'_1\rangle \longrightarrow^* \dots \longrightarrow^* \langle P'_n, c'_n\rangle \longrightarrow^* \langle P'_{n+1}, d\rangle \not\longrightarrow$$

Hence we have $\langle c, d\rangle \in \mathcal{O}(P')$. □

We extend now the definition of simulation to agents and to programs. The first notion will provide the basis for the correctness of the replacement operation. The second notion will allow expressing a sufficient condition for correctness: when a transformed program is reciprocally similar to the original one, then we can be sure that the transformation is correct.

Definition 5 (Simulation and equivalence on agents and programs). *Let D, D' be two programs, and A, A' be two agents. We say that*

- A' simulates A in D, *written* $A \trianglelefteq_D A'$, *iff* $[D, A] \trianglelefteq [D, A']$.
- A' is equivalent to A in D *iff* $A \trianglelefteq_D A'$ *and* $A' \trianglelefteq_D A$.
- D' simulates D, *written* $D \trianglelefteq D'$, *iff for every agent* A, $[D, A] \trianglelefteq [D', A]$.
- D *is* equivalent to D' *iff* $D \trianglelefteq D'$ *and* $D' \trianglelefteq D$. □

Example 6. Consider the following programs

D : { $p(x)$:- $q(x)$	D' : { $p(x)$:- $tell(x = a)$	D'' : { $p(x)$:- $p(x)$.
$q(x)$:- $tell(x = a)$ }	$q(x)$:- $tell(x = a)$ }	$q(x)$:- $tell(x = a)$ }

It is straightforward to check that $[D, p(x)]$, $[D, q(x)]$, $[D', p(x)]$, $[D', q(x)]$, $[D'', q(x)]$ all simulate each other; moreover, they all also simulate $[D'', p(x)]$, while $[D'', p(x)]$ does not simulate any of them. Consequently, $p(x)$ simulates $q(x)$ in D and in D', but not in D'', while $q(x)$ simulates $p(x)$ in all three programs. Finally, D and D' are equivalent to each other and they both simulate D'', while D'' does not simulate D nor D'. □

Simulation satisfies the following properties.

Proposition 7.

1. *The relations $\trianglelefteq$ (on configurations, processes and programs) and $\trianglelefteq_D$ are reflexive and transitive.*
2. *The simulation between agents is preserved by* contexts, *that is, if $A \trianglelefteq_D A'$, then $C[A] \trianglelefteq_D C[A']$ for any context $C[\,]$ (compositionality).* □

Proof

1. Immediate, by reflexivity and transitivity of the entailment relation of the constraint system
2. By case analysis on the various operators. We illustrate here the case of the parallel operator, which is usually the one which causes problems. For the full proof we refer to [2]. Assume that $[D, A] \trianglelefteq [D, A']$. We want to show that for any agent B, $[D, A \parallel B] \trianglelefteq [D, A' \parallel B]$. To this purpose, define

   $$\mathcal{S} = \{(\langle [D, B_1 \parallel B], c_1\rangle, \langle [D, B_2 \parallel B], c_2\rangle) \mid \langle [D, B_1], c_1\rangle \trianglelefteq \langle [D, B_2], c_2\rangle, \\ c_1 \leq c_2 \text{ and } B \in \mathit{Agents}\}$$

   By definition, $(\langle [D, A \parallel B], \mathit{true}\rangle, \langle [D, A' \parallel B], \mathit{true}\rangle) \in \mathcal{S}$. We show now that $\mathcal{S}$ is a simulation. It is easy to see that $\mathcal{S}$ is joint-closed. Further, let $(\langle [D, B_1 \parallel B], c_1\rangle, \langle [D, B_2 \parallel B], c_2\rangle) \in \mathcal{S}$. We need to show that the properties (i) and (ii) of Definition 3 are verified.

   (i) Whenever we have a transition from $\langle [D, B_1 \parallel B], c_1\rangle$, it is either a transition of the form
   (a) $\langle [D, B_1 \parallel B], c_1\rangle \longrightarrow \langle [D, B_1' \parallel B], c_1'\rangle$ (i.e. B_1 makes a step and B is idle), or
   (b) $\langle [D, B_1 \parallel B], c_1\rangle \longrightarrow \langle [D, B_1 \parallel B'], c_1'\rangle$ (i.e. B makes a step and B_1 is idle).

   We consider the two cases separately.
   (a) If it is B_1 which makes the step, then we have also $\langle [D, B_1], c_1\rangle \longrightarrow \langle [D, B_1'], c_1'\rangle$ and, by definition of $\mathcal{S}$, we can derive $\langle [D, B_2], c_2\rangle \longrightarrow^* \langle [D, B_2'], c_2'\rangle$, with $c_1' \leq c_2'$ and $\langle [D, B_1'], c_1'\rangle \trianglelefteq \langle [D, B_2'], c_2'\rangle$. Consequently we have $\langle [D, B_2 \parallel B], c_2\rangle \longrightarrow^* \langle [D, B_2' \parallel B], c_2'\rangle$ with $(\langle [D, B_1' \parallel B], c_1'\rangle, \langle [D, B_2' \parallel B], c_2'\rangle) \in \mathcal{S}$ and $c_1' \leq c_2'$.
   (b) If it is B which makes the step, then let $c_1' = c_1 \wedge c'$ for some suitable c'. Since $c_1 \leq c_2$, we will also have $\langle [D, B_2 \parallel B], c_2\rangle \longrightarrow \langle [D, B_2 \parallel B'], c_2'\rangle$ with $c_2' = c_2 \wedge c'$. By the condition of joint-closedness, $\langle [D, B_1'], c_1'\rangle \trianglelefteq \langle [D, B_2'], c_2'\rangle$ holds. Furthermore $c_1' = c_1 \wedge c' \leq c_2 \wedge c' = c_2'$. Hence $(\langle [D, B_1 \parallel B'], c_1'\rangle, \langle [D, B_2 \parallel B'], c_2'\rangle) \in \mathcal{S}$ and $c_1' \leq c_2'$.

   (ii) If $\langle [D, B_1 \parallel B], c_1\rangle \not\longrightarrow$, then $\langle [D, B_1], c_1\rangle \not\longrightarrow$ and $\langle [D, B], c_1\rangle \not\longrightarrow$. By definition of $\mathcal{S}$ we have that $\langle [D, B_2], c_2\rangle \longrightarrow^* \langle [D, B_2'], c_1\rangle \not\longrightarrow$, from which we derive $\langle [D, B_2 \parallel B], c_2\rangle \longrightarrow^* \langle [D, B_2' \parallel B], c_1\rangle \not\longrightarrow$.

4 The replacement operation and its partial correctness

Given a program D, the replacement operation consists in replacing a number of agents $\{A_1, \ldots, A_n\}$ with new agents $\{A_1', \ldots, A_n'\}$ in the bodies of some of the definitions of D. Here, what we are looking for are conditions sufficient to ensure that the resulting program is equivalent to D. In this section we make the first step in this direction by showing that if each A_i simulates A_i' (in D) then D simulates the program D' resulting from the transformation. In other words, the transformation is *partially correct.*

As mentioned in Proposition 7, the property of "being simulated by" is carried over through context, therefore when we replace A by A' in the body of the definition $p(x)$:- $C[A]$, we can – without loss of generality – look at it as if we were actually replacing its whole body, i.e. $C[A]$ by $C[A']$. This simplifies the notation of the following result, which is the main point of this section and relates the notion of simulation between agents to the notion of simulation between programs.

Theorem 8 (Partial Correctness). *Consider the following ccp programs*

$$D = \{\ p_i(\tilde{x}_i) \text{ :- } A_i\}_{i\in\{1,\ldots,n\}}\ \cup\ E$$
$$D' = \{\ p_i(\tilde{x}_i) \text{ :- } A'_i\}_{i\in\{1,\ldots,n\}}\ \cup\ E$$

If, for each $i \in \{1, \ldots, n\}$, $A'_i \trianglelefteq_D A_i$ *holds, then* $D' \trianglelefteq D$ *holds.*

Proof Let $\mathcal{S}$ be the relation

$$\mathcal{S} = \{(\langle [D', F], c\rangle, \langle [D, H], d\rangle) \mid \langle [D, F], c\rangle \trianglelefteq \langle [D, H], d\rangle\}.$$

In order to prove the thesis it is sufficient to show that $\mathcal{S}$ is a simulation. It is easy to see that $\mathcal{S}$ is joint-closed. We prove now that it satisfies the properties (i) and (ii) of Definition 3.

Let $(\langle [D', F], c\rangle, \langle [D, H], d\rangle)$ be a generic pair in $\mathcal{S}$.

(i) Assume that

$$\langle [D', F], c\rangle \longrightarrow \langle [D', F'], c'\rangle \tag{1}$$

We must prove that there exist H' and d' such that $\langle [D, H], d\rangle \longrightarrow^* \langle [D, H'], d'\rangle$ with $c' \leq d'$ and $(\langle [D', F'], c'\rangle, \langle [D, H'], d'\rangle) \in \mathcal{S}$. We consider two cases, depending on whether the transition (1) contains or not a procedure expansion in the part of D' which is different from D.

1. Assume that (1) contains no procedure expansions, or else they are confined to the E part of D'. In this case the same transition can take place also in D, i.e. we have $\langle [D, F], c\rangle \longrightarrow \langle [D, F'], c'\rangle$. By definition of $\mathcal{S}$ we know that $\langle [D, F], c\rangle \trianglelefteq \langle [D, H], d\rangle$. Hence there exist H', d' such that $\langle [P, H], d\rangle \longrightarrow^* \langle [D, H'], d'\rangle$ with $c' \leq d'$ and $\langle [D, F'], c'\rangle \trianglelefteq \langle [D, H'], d'\rangle$. By definition of $\mathcal{S}$, we have $(\langle [D', F'], c'\rangle, \langle [D, H'], d'\rangle) \in \mathcal{S}$.
2. Assume now that (1) contains a procedure expansion in the subset $\{\ p_i(\tilde{x}_i) \text{ :- } A'_i\}_{i\in\{1,\ldots,n\}}$ of D'. Then F must have the form $C[\ p_i(\tilde{y})\ ]$ where $C[\ ]$ is some context and $i \in \{1, \ldots, n\}$. Hence $F' = C[\Delta^{\tilde{y}}_{\tilde{x}} A'_i]$ and $c' = c$. If we consider the corresponding procedure expansion in D, we have $\langle [D, F], c\rangle \longrightarrow \langle [D, G], c\rangle$, where Let $G = C[\Delta^{\tilde{y}}_{\tilde{x}} A_i]$. Since $A'_i \trianglelefteq_D A_i$, by Proposition 7 we have that $\langle [D, F'], c\rangle \trianglelefteq \langle [D, G], c\rangle$. Finally, by definition of $\mathcal{S}$, we have $\langle [D, F], c\rangle \trianglelefteq \langle [D, H], d\rangle$. Hence there exists H' and d' such that $\langle [D, H], d\rangle \longrightarrow^* \langle [D, H'], d'\rangle$ with $c \leq d'$ and $\langle [D, G], c\rangle \trianglelefteq \langle [D, H'], d'\rangle$. By transitivity of $\trianglelefteq$, we deduce that $\langle [D, F'], c\rangle \trianglelefteq \langle [D, H'], d'\rangle$, which implies $(\langle [D', F'], c\rangle, \langle [D, H'], d'\rangle) \in \mathcal{S}$.

(ii) Assume that $\langle [D', F], c\rangle \not\longrightarrow$. Then $\langle [D, F], c\rangle \not\longrightarrow$ as well. By definition of $\mathcal{S}$ we have $\langle [D, F], c\rangle \trianglelefteq \langle [D, H], d\rangle$. Hence there exists H' such that $\langle [D, H], d\rangle \longrightarrow^* \langle [D, H'], c\rangle \not\longrightarrow$. $\square$

Theorem 8 ensures that if the replacing agents are simulated by the replaced ones then the resulting program is simulated by the original one, i.e. the transformation is *partially* correct. In order to achieve *total correctness* we also have to find conditions which ensure that the resulting program simulates the original one. For this purpose, it would be nice if the converse of the above result would hold, namely, if $A_i \trianglelefteq_D A'_i$ would imply $D \trianglelefteq D'$. Unfortunately this is not the case: Consider again the programs in Example 6, recall that $p(x)$ simulates and is simulated by $q(x)$ in D. Now notice that D'' is the result of replacing $q(x)$ by $p(x)$ in the body of the first definition of D. The fact that D'' is simulated by D, but does not simulate D, shows that the converse of the above theorem does not hold. Hence, in order to obtain applicability conditions for the total correctness of the replacement operation we have to devise new additional tools.

5 Total Correctness by Independence

Example 6 shows a case of program transformation (from D to D'') in which total correctness is not achieved. If we look at the details of the transformation, we note that the replacement of $q(x)$ by $p(x)$ introduces a loop in the program. As we mentioned in the introduction, ensuring that no unwanted loops are brought into the program is the crucial point of ensuring total correctness. In fact, the easiest way to ensure total correctness for the transformation is to require that the definition of the replacing agents does not *depend on* the definitions that are about to be transformed. To formalize this concept, we introduce the following definition.

Definition 9 (Dependency). *Let D be a program, and p and q be predicate symbols. We say that* p refers to q *in D if there is a a definition in D with p in the head and q in the body. We say that* p depends on q *in D if (p, q) is in the reflexive and transitive closure of the relation* refers to. *Finally, we say that an agent A* depends on *p if in A occurs a predicate which depends on p.* $\square$

We can now state our first result on total correctness. Because of space reasons, for its proof we refer to [2].

Theorem 10 (Total Correctness 1). *Consider the following ccp programs*

$$D = \{\, p_i(\tilde{x}_i) \;\text{:-}\; A_i\}_{i\in\{1,\ldots,n\}} \;\cup\; E$$
$$D' = \{\, p_i(\tilde{x}_i) \;\text{:-}\; A'_i\}_{i\in\{1,\ldots,n\}} \;\cup\; E$$

If, for each $i \in \{1 \ldots n\}$, the following two conditions hold:

(1) A_i is equivalent to A'_i in D,
(2) for every $j \in \{1, \ldots, n\}$, A'_i does not depend on p_j.

then D is equivalent to D'. □

In other words, if condition (2) is satisfied, then the converse of Theorem 8 holds. Condition (2) corresponds to the condition (a) mentioned in the introduction, and it ensures, syntactically, that no loops are introduced by the transformation. This confirms our claim that, as long as the replacing agents are equivalent to the replaced ones, if the transformation is not correct then it is only because of the introduction of some unwanted loop.

6 Total Correctness by Improvements

In this section we propose a second method for guaranteeing that no unwanted loops are introduced. While the one seen in Theorem 10 is syntactic (based on Condition (2)), the one we propose here is based on the semantics, and formalizes the requirement (b) mentioned in the introduction. The resulting approach is more complex than the one based on Theorem 10 but it is often more useful for program optimization. The crucial concept here is the one of *improving simulation*, which is a notion of simulation between agents that takes into account the number of procedure expansions in the transitions.

Definition 11 (Improving simulation on configurations). *A relation $\mathcal{S} \subseteq$ Conf $\times$ Conf is an* improving simulation *iff it joint-closed and for every pair $(\langle [D_1, A_1], c_1\rangle, \langle [D_2, A_2], c_2\rangle) \in \mathcal{S}$ the following two conditions hold:*

(i) If $\langle [D_1, A_1], c_1\rangle \longrightarrow_{n_1} \langle [D_1, A_1'], c_1'\rangle$ then for some A_2', c_2', n_2, $\langle [D_2, A_2], c_2\rangle \longrightarrow^*_{n_2} \langle [D_2, A_2'], c_2'\rangle$, $n_1 \geq n_2$, $c_1' \leq c_2'$, and $(\langle [D_1, A_1'], c_1'\rangle, \langle [D_2, A_2'], c_2'\rangle) \in \mathcal{S}$.
(ii) If $\langle [D_1, A_1], c_1\rangle \not\longrightarrow$, then for some A_2', $\langle [D_2, A_2], c_2\rangle \longrightarrow^*_0 \langle [D_2, A_2'], c_1\rangle \not\longrightarrow$. □

As a notational convention, if $(\langle P, c\rangle, \langle P', c'\rangle) \in \mathcal{S}$ for some improving simulation $\mathcal{S}$, then we say that $\langle P', c'\rangle$ *improves on* $\langle P, c\rangle$ and we write $\langle P, c\rangle \preceq \langle P', c'\rangle$. Again, we further say that the process P' *improves on* the process P ($P \preceq P'$) iff $\langle P, \mathit{true}\rangle \preceq \langle P', \mathit{true}\rangle$.

Now, in order to present the other results concerning the total correctness of the replacement operation, we need to extend the concept of improving simulation to agents and programs.

Definition 12 (Improvement relation on agents and programs). *Let D, D' be two programs, and let A, A' be two agents. We say that:*

- A' improves on A in D, *written $A \preceq_D A'$, iff $[D, A] \preceq [D, A']$.*
- D' improves on D *iff, for every agent A, $[D, A] \preceq [D', A]$.* □

The relation of improvement is strictly more restrictive than the one of simulation: if an agent (resp. process, program) *improves on* another one then it certainly simulates it as well. Of course, the converse is not true. Consider again the programs in Example 6. It is straightforward to check that:

- In D, $q(x)$ improves on $p(x)$, but not vice-versa;
- In D', $q(x)$ improves on $p(x)$, and vice-versa;
- In D'', $q(x)$ improves on $p(x)$ but not vice-versa ($p(x)$ does not even simulate $q(x)$ in D'').

Furthermore, we also have that D' improves on D, while D simulates D' but it is not an improvement on it. In fact in D' the agent $p(x)$ after a single procedure expansion performs the action *tell*, while in D the same agent must perform two procedure expansions before performing the *tell* action.

Proposition 7 applies to the improvement relation as well: all improvement relations are reflexive and transitive. Moreover, the improvement relation between agents is invariant under contexts.

We are now ready to state our main result.

Theorem 13 (Total Correctness 2). *Consider the following ccp programs*

$$D = \{\, p_i(\tilde{x}_i) \text{ :- } A_i\}_{i\in\{1,\ldots,n\}} \ \cup\ E$$
$$D' = \{\, p_i(\tilde{x}_i) \text{ :- } A'_i\}_{i\in\{1,\ldots,n\}} \ \cup\ E$$

1. *If, for each $i \in \{1,\ldots,n\}$, $A_i \preceq_D A'_i$ holds, then D' improves on D.*
2. *If, in addition, for each $i \in \{1,\ldots,n\}$, $A'_i \trianglelefteq_D A_i$, then D' is equivalent to D.*

Proof We prove only Part (1). Part (2) is a consequence of the combination of Part (1) and Theorem 8.

Let $\mathcal{S}$ be the relation

$$\mathcal{S} = \{(\langle [D,F],c\rangle, \langle [D',H],d\rangle) \mid \langle [D,F],c\rangle \ \preceq\ \langle [D,H],d\rangle\}.$$

In order to prove the thesis it is sufficient to show that $\mathcal{S}$ is an improving simulation. It is easy to see that $\mathcal{S}$ is joint-closed. We prove now that it satisfies the properties (i) and (ii) of Definition 11.

Let $(\langle [D,F],c\rangle, \langle [D',H],d\rangle)$ be a generic pair in $\mathcal{S}$.

(i) Assume that $\langle [D,F],c\rangle \longrightarrow_n \langle [D,F'],c'\rangle$, where $n=0$ or $n=1$. We must prove that there exist H' and d' such that $\langle [D',H],d\rangle \longrightarrow^*_m \langle [D',H'],d'\rangle$ with $m \le n$, $c' \le d'$, and $(\langle [D,F'],c'\rangle, \langle [D',H'],d'\rangle) \in \mathcal{S}$.
By definition of $\mathcal{S}$ we have $\langle [D,F],c\rangle \ \preceq\ \langle [D,H],d\rangle$, hence there exist H', m and d' such that

$$\langle [D,H],d\rangle \longrightarrow^*_k \langle [D,H'],d'\rangle \tag{2}$$

with $m \le n$, $c' \le d'$, and $\langle [D,F'],c'\rangle \ \preceq\ \langle [D,H'],d'\rangle$.
We consider two cases, depending on whether the transition sequence (2) contains or not a procedure expansion in the part of D' which is different from D.

1. Assume that (2) contains no procedure expansions, or else they are confined to the E part of D'. In this case the same transitions can take place also in D', i.e. we have $\langle [D',H],d\rangle \longrightarrow^*_k \langle [D',H'],d'\rangle$. Furthermore, by definition of $\mathcal{S}$, we have $(\langle [D,F'],c'\rangle, \langle [D',H'],d'\rangle) \in \mathcal{S}$.

2. Assume now that (2) contains a procedure expansion in the subset $\{\ p_i(\tilde{x}_i)\ \text{:-}\ A'_i\}_{i\in\{1,\ldots,n\}}$ of D'. Since m can only be 1, then (2) must be of the form

   $$\langle [D,H],d\rangle \longrightarrow_0^* \langle [D,G],e\rangle \longrightarrow_1 \langle [D,L],e\rangle \longrightarrow_0^* \langle [D,H'],d'\rangle$$

   where $G = C[p_i(\tilde{y}_i)]$ and $L = C[\Delta_{\tilde{x}_i}^{\tilde{y}_i} A_i]$. If we replace D by D' we obtain

   $$\langle [D',H],d\rangle \longrightarrow_0^* \langle [D',G],e\rangle \longrightarrow_1 \langle [D',L'],e\rangle$$

   where $L' = C[\Delta_{\tilde{x}_i}^{\tilde{y}_i} A'_i]$. Furthermore, since $A_i \preceq_D A'_i$, by Proposition 7 we have that $\langle [D,L],e\rangle\ \preceq\ \langle [D,L'],e\rangle$. Hence

   $$\langle [D,L'],e\rangle \longrightarrow_0^* \langle [D,H''],d''\rangle$$

   with $d' \leq d''$ and $\langle [D,H'],d'\rangle\ \preceq\ \langle [D,H''],d''\rangle$. Since there are no procedure expansions, the same sequence of transitions can be obtained in D':

   $$\langle [D',L'],e\rangle \longrightarrow_0^* \langle [D',H''],d''\rangle$$

Finally, observe that, by transitivity of $\preceq$ we have $\langle [D,F'],c'\rangle \preceq \langle [D,H''],d''\rangle$. Hence, by definition of $\mathcal{S}$, we obtain $(\langle [D,F'],c'\rangle, \langle [D',H''],d''\rangle) \in \mathcal{S}$.

(ii) Assume that $\langle [D,F],c\rangle \not\longrightarrow$. By definition of $\mathcal{S}$, $(\langle [D,F],c\rangle, \langle [D,H],d\rangle) \in \mathcal{S}$, hence there exists H' such that $\langle [D,H],d\rangle \longrightarrow_0^* \langle [D,H'],c\rangle \not\longrightarrow$. Since there are no procedure expansions, the same sequence of transitions can be obtained in D': $\langle [D',H],d\rangle \longrightarrow_0^* \langle [D',H'],c\rangle \not\longrightarrow$. $\square$

Two remarks are in order: First, when we apply the replacement operation in order to improve the efficiency of a program it is natural to require that the replacing agent be equivalent to and more efficient than the replaced one. The above theorem shows the pleasing properties that under those circumstances the replacement is always correct and that it yields a program which is an improvement on the initial one. Secondly, from Proposition 4 it follows that, when the above theorem applies, then also the semantics of the observables is preserved, i.e. for any agent A, $\mathcal{O}([D',A]) = \mathcal{O}([D,A])$ holds.

7 An extended example: minimum and maximum element of a list

In order to illustrate some of the possible uses of the applicability conditions formulated above, we use here a typical example of an unfold-fold transformation system. Let D be the following ccp program

```
min(list,m) :- (   ask (∃x,w,xs list = [x,w|xs]) → ∃z,x,xs (tell(list = [x|xs]) || min(xs,z)
                                                              || smaller(x,z,m)              )
                 + ask (∃x list = [x]) → ∃x (tell(list = [x]) || tell(m = x)))

max(list,m) :- (   ask (∃x,w,xs list = [x,w,|xs]) → ∃z,x,xs (tell(list = [x|xs]) || max(xs,z)
                                                              || greater(x,z,m)               )
                 + ask (∃x list = [x]) → ∃x (tell(list = [x]) || tell(m = x)))

minmax(l,min,max) :- min(l,min) || max(l,max)
```

Where $\mathsf{smaller}$ and $\mathsf{greater}$ are defined in the obvious way. Here $\mathsf{min(list,m)}$ returns in m the minimum value of the non-empty list list: in the first branch of the ask choice, min checks if list contains more than one element, in which case is call itself recursively. On the other hand, if list contains only one element x, then the second branch is followed and m is set equal to x. max works in the same way, and minmax reports both the minimum and the maximum of the values in the list. Notice that the definition of minmax traverses the input list twice. This is a source of inefficiency, which can be fixed via an unfold/fold transformation. The first operation we encounter is the *unfolding*, which consists of replacing an atom with the body of its definition. Consider the program

$$
\begin{array}{rll}
D \;=\; \{ & d_1: & q(\tilde{y}) \text{ :- } C[p(\tilde{v})] \\
 & d_2: & p(\tilde{x}) \text{ :- } A \qquad\quad \} \cup E
\end{array}
$$

then *unfolding* $p(\tilde{v})$ in d_1 means replacing d_1 with $d_1' :\ q(\tilde{y}) \text{ :- } C[\Delta^{\tilde{v}}_{\tilde{x}} A]$. This operation can be regarded as an instance of replacement. In fact, its correctness follows in a straightforward manner from Theorem 13: it is easy to show that in the above program $\Delta^{\tilde{v}}_{\tilde{x}} A$ is *equivalent to*, and *improves on* $p(\tilde{v})$. By applying twice the unfolding operation to the definition of minmax, we obtain the following definition.

$$
\begin{array}{l}
\mathsf{minmax(l,min,max)\ \text{:-}} \\
\qquad (\ \ \mathsf{ask}\ (\exists_{x,w,xs}\ \mathsf{l = [x,w|xs]}) \rightarrow \exists_{z,x,xs}\ (\mathsf{tell(l = [x|xs])} \parallel \mathsf{min(xs,z)} \parallel \mathsf{smaller(x,z,min)}) \\
\qquad +\ \mathsf{ask}\ (\exists_{x}\ \mathsf{l = [x]}) \rightarrow \exists_{x}\ (\mathsf{tell(l = [x])} \parallel \mathsf{tell(min = x)})) \\
\quad \parallel \\
\qquad (\ \ \mathsf{ask}\ (\exists_{x',w',xs'}\ \mathsf{l = [x',w',|xs']}) \rightarrow \\
\qquad\qquad\qquad \exists_{z',x',xs'}\ (\ \mathsf{tell(l = [x'|xs'])} \parallel \mathsf{max(xz',z')} \parallel \mathsf{greater(x',z',max)}\ ) \\
\qquad +\ \mathsf{ask}\ (\exists_{x'}\ \mathsf{l = [x']}) \rightarrow \exists_{x'}\ (\mathsf{tell(l = [x'])} \parallel \mathsf{tell(max = x')}))
\end{array}
$$

Now, it is straightforward to prove that if $a \wedge b = false$, then $(((ask(a) \rightarrow A) + (ask(b) \rightarrow B)) \parallel ((ask(a) \rightarrow C) + (ask(b) \rightarrow D)))$ improves on $(ask(a) \rightarrow (A \parallel C)) + (ask(b) \rightarrow (B \parallel D))$ and vice-versa. Thus, by Theorem 13, we can safely apply a replacement operation which yields

$$
\begin{array}{l}
\mathsf{minmax(l,min,max)\ \text{:-}} \\
\qquad \mathsf{ask}\ (\exists_{x,w,xs}\ \mathsf{l = [x,w|xs]}) \rightarrow \\
\qquad\qquad (\quad \exists_{z,x,xs}\ (\mathsf{tell(l = [x|xs])} \parallel \mathsf{min(xs,z)} \parallel \mathsf{smaller(x,z,min)}) \\
\qquad\qquad \parallel \quad \exists_{z',x',xs'}\ (\mathsf{tell(l = [x'|xs'])} \parallel \mathsf{max(xs',z')} \parallel \mathsf{greater(x',z',max)})) \\
\quad +\ \mathsf{ask}\ (\exists_{x}\ \mathsf{l = [x]}) \rightarrow (\quad \exists_{x}\ (\mathsf{tell(l = [x])} \parallel \mathsf{tell(min = x)})) \\
\qquad\qquad\qquad\qquad\quad \parallel \quad \exists_{x'}\ (\mathsf{tell(l = [x'])} \parallel \mathsf{tell(max = x')})
\end{array}
$$

Now, with other (intuitively immediate) replacement operations whose correctness is guaranteed by Theorem 13, we obtain the following definition.

$$
\begin{array}{l}
\mathsf{minmax(l,min,max)\ \text{:-}} \\
\qquad \mathsf{ask}(\exists_{x,w,xs}\ \mathsf{l = [x,w|xs]}) \rightarrow \exists_{z,z',x,xs}\ (\ \mathsf{tell(l = [x|xs])} \parallel \mathsf{smaller(x,z,min)} \parallel \mathsf{min(xs,z)} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \parallel \mathsf{max(xs,z')} \parallel \mathsf{greater(x,z',max)}) \\
\quad +\ \mathsf{ask}(\exists_{x}\ \mathsf{l = [x]}) \rightarrow \exists_{x}\ (\mathsf{tell(min = x)} \parallel \mathsf{tell(max = x)} \parallel \mathsf{tell(l = [x])})
\end{array}
$$

We are now ready for the last operation, which corresponds to a folding one: we replace $\mathsf{min(xs,z)} \parallel \mathsf{max(xs,z')}$ with $\mathsf{minmax(xs,z,z')}$.

```
minmax(l,min,max) :-
        ask(∃_{x,w,xs} l = [x,w|xs]) → ∃_{z,z',x,xs} (tell(l = [x|xs]) || minmax(xs,z,z')
                                             || smaller(x,z,min) || greater(x,z',max))
   + ask(∃_x l = [x]) → ∃_x (tell(l = [x]) || tell(min = x) || tell(max = x))
```

Notice that this operation has made *recursive* the definition of minmax. As we mentioned in the introduction, this is an example of a *wanted loop*. Clearly, for this last operation we could have not applied Theorem 10

As the mimmax example shows, the applicability conditions we propose for the replacement operation are flexible enough to let this operation mimic both unfolding, folding and usual (yet not trivial) "cleaning-up" operations. (See Appendix A for a list of agents which can be interchanged). Actually, it is possible to devise a fold-unfold transformation system whose proof of correctness is based on Theorem 13. This has been done in [2]. Such a system is not quite as powerful as a tailored folding operation such the one presented in [9] (which, on the other hand, ensure only a weaker form of correctness of the transformation), but this accounts for the flexibility of the conditions we presented.

8 Concluding Remarks

For experts in transformations, the applicability conditions outlined in Theorem 13 are not fully surprising: similar concepts were first employed in Logic Programming (for instance in [28, 4, 5, 7]) and then applied to functional programs in [24, 25]. However, this is the first time that they are applied to a concurrent context. To the best of our knowledge, this is the first paper which treats the notion of replacement in a concurrent language.

Acknowledgements We would like to thank the anonymous referees of LOPSTR and of APPIA-GULP-PRODE for the useful suggestions.

References

1. N. Bensaou and I. Guessarian. Transforming constraint logic programs. *Theoretical Computer Science*, 206(1–2):81–125, 1998.
2. M. Bertolino. Transformazione dei programmi concorrenti Tesi di Laurea, Dip. Informatica e Scienze dell' Informazione, Università di Genova, Genova, Italy, 1997.
3. A. Bossi and N. Cocco. Basic Transformation Operations which preserve Computed Answer Substitutions of Logic Programs. *Journal of Logic Programming*, 16(1&2):47–87, 1993.
4. A. Bossi, N. Cocco, and S. Etalle. On Safe Folding. In M. Bruynooghe and M. Wirsing, editors, *Programming Language Implementation and Logic Programming - Proceedings PLILP'92*, volume 631 of *Lecture Notes in Computer Science*, pages 172–186. Springer-Verlag, 1992.
5. A. Bossi, N. Cocco, and S. Etalle. Simultaneous replacement in normal programs. *Journal of Logic and Computation*, 6(1):79–120, February 1996.

6. J. Cook and J.P. Gallagher. A transformation system for definite programs based on termination analysis. In F. Turini, editor, *Proc. Fourth Workshop on Logic Program Synthesis and Transformation*. Springer-Verlag, 1994.
7. S. Etalle and M. Gabbrielli. On the correctness of the replacement operation for clp modules. *Journal of Functional and Logic Programming*, February 1996. available at http://www.cs.tu-berlin.de/journal/jflp.
8. S. Etalle and M. Gabbrielli. Transformations of CLP modules. *Theoretical Computer Science*, 166(1):101–146, 1996.
9. S. Etalle, M. Gabbrielli, and M. C. Meo. Unfold/Fold Transformations of CCP Programs. In D. Sangiorgi and R. de Simone, editors, *CONCUR98 – 1998 International Conference on Concurrency Theory*, LNCS 1466, pages 348–363. Springer-Verlag, 1998.
10. N. De Francesco and A. Santone. Unfold/fold transformation of concurrent processes. In H. Kuchen and S.Doaitse Swierstra, editors, *Proc. 8th Int'l Symp. on Programming Languages: Implementations, Logics and Programs*, volume 1140, pages 167–181. Springer-Verlag, 1996.
11. H. Fujita, A. Okumura, and K. Furukawa. Partial evaluation of GHC programs based on the UR-set with constraints. In R.A. Kowalski and K.A. Bowen, editors, *Logic Programming: Fifth International Conference and Symposium, volume 2*, pages 924–941. Cambridge, MA: MIT Press, 1988.
12. P.A. Gardner and J.C. Shepherdson. Unfold/fold transformations of logic programs. In J-L Lassez and G. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*. MIT Press, 1991.
13. M. Gengler and M. Martel. Self-applicable partial evaluation for the pi-calculus. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '97)*, pages 36–46. ACM, 1997.
14. H. Hosoya, N. Kobayashi, and A. Yonezawa. Partial evaluation scheme for concurrent languages and its correctness. In L. Bougé et al., editors, *Euro-Par'96 - Parallel Processing, Lyon, France. (Lecture Notes in Computer Science, vol. 1123)*, pages 625–632. Berlin: Springer-Verlag, 1996.
15. Niels Jørgensen, Kim Marriott, and Spiro Michaylov. Some global compile-time optimizations for CLP($\mathcal{R}$). In Vijay Saraswat and Kazunori Ueda, editors, *International Logic Programming Symposium*, pages 420–434, San Diego, 1991. MIT Press.
16. M.J. Maher. A transformation system for deductive databases with perfect model semantics. *Theoretical Computer Science*, 110(2):377–403, March 1993.
17. M. Marinescu and B. Goldberg. Partial-evaluation techniques for concurrent programs. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '97)*, pages 47–62. ACM, 1997.
18. Kim Marriott and Harald Søndergaard. Analysis of constraint logic programs. In Saumya Debray and Manuel Hermenegildo, editors, *Proceedings North American Conference on Logic Programming*. MIT Press, 1990.
19. Kim Marriott and Peter J. Stuckey. The 3 r's of optimizing constraint logic programs: Refinement, removal and reordering. In *POPL'93: Proceedings ACM SIGPLAN Symposium on Principles of Programming Languages*, Charleston, January 1993.
20. D.M.R. Park. Concurrency and automata on infinite sequences. In P. Deussen, editor, *Proc. of the 5th GI conference*, Lecture Notes in Computer Science, pages 167–183. Springer-Verlag, 1981.
21. A. Pettorossi and M. Proietti. Transformation of logic programs: Foundations and techniques. *Journal of Logic Programming*, 19,20:261–320, 1994.

22. M. Proietti and A. Pettorossi. Synthesis and transformation of logic programs using unfold/fold proofs. *Journal of Logic Programming*, 41(2-3):197–230, 1999.
23. D. Sahlin. Partial Evaluation of AKL. In *Proceedings of the First International Conference on Concurrent Constraint Programming*, 1995.
24. D. Sands. Total correctness by local improvement in program transformation. In *Proceedings of the 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM Press, 1995.
25. D. Sands. Total correctness by local improvement in the transformation of functional programs. *ACM Transactions on Programming Languages and Systems*, 18(2):175–234, 1996.
26. V.A. Saraswat, M. Rinard, and P. Panangaden. Semantics foundations of concurrent constraint programming. In *Proc. Eighteenth Annual ACM Symp. on Principles of Programming Languages*. ACM Press, 1991.
27. T. Sato. Equivalence-preserving first-order unfold/fold transformation system. *Theoretical Computer Science*, 105(1):57–84, 1992.
28. H. Tamaki and T. Sato. Unfold/Fold Transformations of Logic Programs. In Sten-Åke Tärnlund, editor, *Proc. Second Int'l Conf. on Logic Programming*, pages 127–139, 1984.
29. K. Ueda and K. Furukawa. Transformation rules for GHC Programs. In *Proc. Int'l Conf. on Fifth Generation Computer Systems*, pages 582–591. Institute for New Generation Computer Technology, Tokyo, 1988.
30. R.J. van Glabbeek. The linear time - branching time spectrum. In J.C.M. Baeten and J.W. Klop, editors, *Proc. of CONCUR 90*, volume 458 of *Lecture Notes in Computer Science*, pages 278–297, Amsterdam, 1990. Springer-Verlag.

A Appendix: Mutually replaceable Agents

In this appendix we present a list of mutually replaceable agents, that is, agents which, under the reported applicability conditions, are equivalent to each others (in all programs) and which improve on each other (in all programs). In virtue of Theorem 13 these agents can (under the given conditions) be freely interchanged. Hence the name "mutually replaceable". The technical proofs are presented in [2, Appendix 2].

Here, for the sake of simplicity, we write $A \equiv B$ as a shorthand for "A is mutually replaceable with B".

Properties of the parallel operator

- $A \parallel stop \equiv A$
- $A \parallel B \equiv B \parallel A$
- $A \parallel (B \parallel C) \equiv (A \parallel B) \parallel C$

Properties of the tell operator

- $tell(true) \equiv stop$
- $tell(c) \parallel tell(d) \equiv tell(c \wedge d)$

Properties of the ask operator

- $ask(true) \to A \equiv A$
- $ask(c) \to (ask(d) \to A) \equiv ask(c \wedge d) \to A$

Properties of the hiding operator

- $\exists_x A \equiv A$ provided that $x \notin FV(A)$
- $\exists_{\tilde{x}} A \parallel \exists_{\tilde{x}} B \equiv \exists_{\tilde{x}} (A \parallel \exists_{\tilde{x}} B)$
- $\exists_{\tilde{x}} \exists_{\tilde{y}} A \equiv \exists_{\tilde{y}} \exists_{\tilde{x}} A$
- $\exists^d_{\tilde{x}} \exists^e_{\tilde{y}} A \equiv \exists^e_{\tilde{y}} \exists^d_{\tilde{x}} A$ provided that $\exists_{\tilde{y}} d = d$, $\exists_{\tilde{x}} e = e$
- $\exists^d_{\tilde{x}} \exists^e_{\tilde{y}} A \equiv \exists^{d \wedge e}_{\tilde{x}\tilde{y}} A$ provided that $\exists_{\tilde{y}} d = d$
- $\exists^{c \wedge e}_{\tilde{x}} \exists^d_{\tilde{y}} A \equiv \exists^c_{\tilde{x}} \exists^{e \wedge d}_{\tilde{y}} A$ provided that $\exists_{\tilde{y}} e = e$

Mixed properties

- $\exists_{\tilde{x}} tell(c) \equiv tell(\exists_{\tilde{x}} c)$
- $\exists_{\tilde{x}} (tell(c) \parallel A) \equiv tell(\exists_{\tilde{x}} c) \parallel \exists^c_{\tilde{x}} A$
- $\exists_{\tilde{x}} \sum_{i=1}^n ask(c_i) \to A_i \equiv \sum_{i=1}^n ask(c_i) \to \exists_{\tilde{x}} A_i$ provided that $\exists_{\tilde{x}} c_i = c_i$ for every $i \in \{1, \ldots, n\}$
- $((ask(a) \to A + ask(b) \to B) \parallel (ask(a) \to C + ask(b) \to D)) \equiv ((ask(a) \to (A \parallel C)) + (ask(b) \to (B \parallel D)))$ provided that $a \wedge b = false$
- $tell(d) \parallel \sum_{i=1}^n ask(c_i) \to A_i \equiv tell(d) \parallel A_j$ provided that $c_j \leq d$ and for every $k \in \{1, \ldots, n\}, k \neq j \Rightarrow d \wedge c_k = false$