DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN
UNIVERSIDAD POLITÉCNICA DE VALENCIA

P.O. Box: 22012 E-46071 Valencia (SPAIN)

Informe Tcnico / Technical Report

Ref. No.: DSIC-II/03/09 **Pages:** 27
Title: A Tracing Semantics for CSP
Author(s): M. Llorens, J. Oliver, J. Silva and S. Tamarit
Date: June, 2009
Keywords: Concurrent Programming, Semantics, Tracing, CSP

Vº Bº
Leader of research Group Author(s)

A Tracing Semantics for CSP*

Marisa Llorens, Javier Oliver, Josep Silva, and Salvador Tamarit

Universidad Politécnica de Valencia, Camino de Vera S/N, E-46022 Valencia, Spain.
{mllorens,fjoliver,jsilva,stamarit}@dsic.upv.es

Abstract. CSP is a powerful language to specify complex concurrent systems. Due to the non-deterministic execution order of processes and to the restrictions imposed on this order by synchronizations, many analyses such as deadlock analysis, reliability analysis, program slicing and the MEB and CEB analyses try to predict properties of the specification which can guarantee the quality of the final system. These analyses often rely on the use of traces and/or on a data structure which is able to represent computations. In this work, we introduce the theoretical basis for tracing concurrent and explicitly synchronized computations in process algebras such as CSP. Tracing computations is a difficult task due to the subtleties of the underlying operational semantics which combines concurrency, non-determinism and non-termination. We define an instrumented operational semantics that generates as a side-effect an appropriate data structure which can be used to trace computations at an adequate level of abstraction. The formal definition of a tracing semantics improves the understanding of the tracing process, but also, it allows us to formally prove the correctness of the computed traces.

Keywords: Concurrent Programming, Semantics, Tracing, CSP.

1 Introduction

One of the most important techniques for program understanding and debugging is tracing. A trace gives the user access to otherwise invisible information about a computation. In the context of concurrent languages, computations are particularly complex due to the non-deterministic execution order of processes and to the restrictions imposed on this order by synchronizations; and thus, a tracer is a powerful tool to explore, understand and debug concurrent computations.

One of the most extended concurrent languages is the *Communicating Sequential Processes* (CSP) [3, 10] whose operational semantics allows the combination of parallel, non-deterministic and non-terminating processes. The study and transformation of CSP specifications often implies different analyses such as deadlock analysis [5], reliability analysis [4] and program slicing [11] which are often based on a data structure able to represent computations.

* This work has been partially supported by the Spanish *Ministerio de Ciencia e Innovación* under grant TIN2008-06622-C03-02, by the *Generalitat Valenciana* under grant GVPRE/2008/001, and by the *Universidad Politécnica de Valencia* (Programs PAID-05-08 and PAID-06-08).

However, computing CSP traces is a complex task due to the non-deterministic execution of processes, due to deadlocks, due to non-terminating processes and mainly due to synchronizations. This is probably the reason why there does not exist any correctness result which formally relates the trace of a specification to the execution of this specification. This semantics is needed because it would allow us to prove important properties (such as correctness and completeness) of the techniques and tools based on tracing.

In this work, we introduce the first tracing semantics for CSP. Concretely, we instrument the standard operational semantics of CSP in such a way that the execution of the semantics produces as a side-effect the trace of the computation. If the execution is stopped (e.g., because it is non-terminating), the semantics produces the trace of the computation performed so far. This semantics can serve as a theoretical foundation for tracing CSP because it formally relates the computations of the standard semantics with the traces of these computations. Therefore, it allows us to better understand the overall tracing process, but it also allows us to formally prove the correctness of the computed traces.

It is important to remark the difference between our traces which are based on the standard notion of trace used in programming languages, and the notion of trace commonly used in CSP which refer to a sequence of events. Concretely, the operational semantics of CSP is an event-based semantics in which the occurrence of events fires the rules of the semantics. Hence, the final trace of the computation is the sequence of events occurred (see Chapter 8 of [10] for a detailed study of this kind of traces). Our notion of trace is essentially different. In our setting, a trace [1, 2] is a data structure which represents the sequence of expressions that have been reduced during the computation, and moreover, this data structure is labeled with the source position of these expressions in the program.

The rest of the paper has been organized as follows. Firstly, in Section 2 we present the standard operational semantics of CSP. In Section 3 we introduce some notation that will be used along the paper. Then, in Section 4, we instrument the semantics in such a way that the execution of the instrumented semantics produces as a side-effect the trace associated to the performed computation. In Section 5, we present the main results of the paper stating that the instrumented semantics presented is a conservative extension of the standard semantics, and its computed traces are correct. Finally, Section 6 concludes.

2 The semantics of CSP

This section presents the standard operational semantics of CSP as defined by Roscoe [10]. It is presented in Fig. 1 as a logical inference system. A *state* of the semantics is an expression to be evaluated called the *control*. The system starts with an initial state, and the rules of the semantics are used to infer how this state evolves. When no rules can be applied to the current state, the computation finishes. The rules of the semantics change the states of the computation due to the occurrence of events. The set of possible events is $\Sigma \cup \{\tau, \checkmark\}$. Events in

(Process Call)

$$\frac{}{P \xrightarrow{\tau} rhs(P)}$$

(Prefixing)

$$\frac{}{(a \rightarrow P) \xrightarrow{a} P}$$

(SKIP)

$$\frac{}{\texttt{SKIP} \xrightarrow{\checkmark} \top}$$

(Internal Choice 1)

$$\frac{}{(P \sqcap Q) \xrightarrow{\tau} P}$$

(Internal Choice 2)

$$\frac{}{(P \sqcap Q) \xrightarrow{\tau} Q}$$

(External Choice 1)

$$\frac{P \xrightarrow{\tau} P'}{(P \,\Box\, Q) \xrightarrow{\tau} (P' \,\Box\, Q)}$$

(External Choice 2)

$$\frac{Q \xrightarrow{\tau} Q'}{(P \,\Box\, Q) \xrightarrow{\tau} (P \,\Box\, Q')}$$

(External Choice 3)

$$\frac{P \xrightarrow{a \text{ or } \checkmark} P'}{(P \,\Box\, Q) \xrightarrow{a \text{ or } \checkmark} P'}$$

(External Choice 4)

$$\frac{Q \xrightarrow{a \text{ or } \checkmark} Q'}{(P \,\Box\, Q) \xrightarrow{a \text{ or } \checkmark} Q'}$$

(Synchronized Parallelism 1)

$$\frac{P \xrightarrow{a \text{ or } \{\tau \text{ or } \checkmark\}} P'}{(P \underset{X}{\parallel} Q) \xrightarrow{a \text{ or } \tau} (P' \underset{X}{\parallel} Q)}\ a \notin X$$

(Synchronized Parallelism 2)

$$\frac{Q \xrightarrow{a \text{ or } \{\tau \text{ or } \checkmark\}} Q'}{(P \underset{X}{\parallel} Q) \xrightarrow{a \text{ or } \tau} (P \underset{X}{\parallel} Q')}\ a \notin X$$

(Synchronized Parallelism 3)

$$\frac{P \xrightarrow{a} P' \quad Q \xrightarrow{a} Q'}{(P \underset{X}{\parallel} Q) \xrightarrow{a} (P' \underset{X}{\parallel} Q')}\ a \in X$$

(Synchronized Parallelism 4)

$$\frac{}{(\top \underset{X}{\parallel} \top) \xrightarrow{\checkmark} \top}$$

(Sequential Composition 1)

$$\frac{P \xrightarrow{a \text{ or } \tau} P'}{(P;Q) \xrightarrow{a \text{ or } \tau} (P';Q)}$$

(Sequential Composition 2)

$$\frac{P \xrightarrow{\checkmark} \top}{(P;Q) \xrightarrow{\tau} Q}$$

(Hiding 1)

$$\frac{P \xrightarrow{a} P'}{(P\backslash B) \xrightarrow{\tau} (P'\backslash B)}\ a \in B$$

(Hiding 2)

$$\frac{P \xrightarrow{a \text{ or } \tau} P'}{(P\backslash B) \xrightarrow{a \text{ or } \tau} (P'\backslash B)}\ a \notin B$$

(Hiding 3)

$$\frac{P \xrightarrow{\checkmark} \top}{(P\backslash B) \xrightarrow{\checkmark} \top}$$

(Renaming 1)

$$\frac{P \xrightarrow{a} P'}{(P[\![R]\!]) \xrightarrow{b} (P'[\![R]\!])}\ a\ R\ b$$

(Renaming 2)

$$\frac{P \xrightarrow{a \text{ or } \tau} P'}{(P[\![R]\!]) \xrightarrow{a \text{ or } \tau} (P'[\![R]\!])}\ a \notin R$$

(Renaming 3)

$$\frac{P \xrightarrow{\checkmark} \top}{(P[\![R]\!]) \xrightarrow{\checkmark} \top}$$

Fig. 1. CSP's operational semantics

$\Sigma = \{a, b, c \ldots\}$ are visible from the external environment, and can only happen with its co-operation (e.g., actions of the user). The special event τ cannot be observed from outside the system and it happens automatically as defined by the

semantics. ✓ is a special event representing the successful termination of a process. We use the special symbol $\top$ to denote any process that already terminated. In order to perform computations, we construct an initial state (e.g., `MAIN`) and (non-deterministically) apply the rules of Fig. 1. The intuitive meaning of each rule is the following:

(Process Call) The call is unfolded and the right-hand side of process P is added to the control. For concretion, in the following, we will use $rhs(P)$ to denote the right-hand side expression in the definition of process P.

(Prefixing) When event a occurs, process P is added to the control. This rule is used both for prefixing and communication operators (input and output). Given a communication expression, either $c?u \rightarrow P$ or $c!u \rightarrow P$, this rule treats the expression as a prefixing except for the fact that the set of u's appearing in P is replaced by the communicated events.

(SKIP) After `SKIP`, the only possible event is ✓, which denotes the end of the (sub)computation with the special symbol $\top$. There is no rule for $\top$ (neither for `STOP`), hence, this (sub)computation has finished.

(Internal Choice 1 and 2) The system uses the internal event τ to (non-deterministically) select one of the two processes P or Q which is added to the control.

(External Choice 1, 2, 3 and 4) The occurrence of τ develops one of the branches. The occurrence of an event $a \neq \tau$ is used to select one of the two processes P or Q and the control changes according to the event.

(Synchronized Parallelism 1 and 2) When event $a \notin X$ or events τ or ✓ happen, one of the two processes P or Q evolves accordingly, but only a is visible from outside the parallelism operator.

(Synchronized Parallelism 3) When event $a \in X$ happens, it is required that both processes synchronize, P and Q are executed at the same time and the control becomes $P' \underset{X}{\parallel} Q'$.

(Synchronized Parallelism 4) When both processes have successfully terminated the control becomes $\top$, performing ✓.

(Sequential Composition 1) In $P;Q$, P can evolve to P' with any event except ✓. Hence, the control becomes $P';Q$.

(Sequential Composition 2) When P finishes (with event ✓), Q starts. Note that ✓ is hidden from outside the whole process becoming τ.

(Hiding 1) When event $a \in B$ occurs in P, it is hidden, and thus changed to τ so that it is not observable from outside P.

(Hiding 2 and Hiding 3) P can normally evolve (using rule 2) until it is finished (✓ happens). When P finishes, the control becomes $\top$.

(Renaming 1) Whenever a renamed event a happens in P, it is renamed to b so that, externally, only b is visible.

(Renaming 2 and 3) Renaming has no effect on either τ or ✓ events. The rules for renaming are similar to those for hiding.

We illustrate the semantics with the following example.

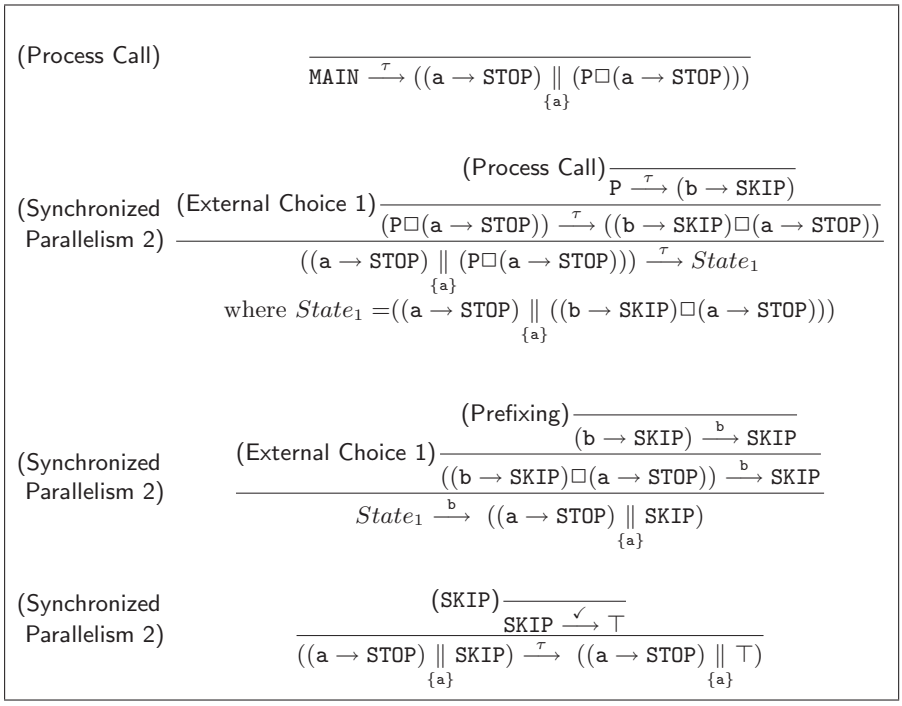


Fig. 2. A computation with the operational semantics in Fig. 1

Example 1. Consider the following CSP specification:

```
MAIN = (a → STOP) ||{a} (P □ (a → STOP))
P = b → SKIP
```

If we use `MAIN` as the initial state to execute the semantics, we get the computation shown in Fig. 2 where the final state is ((a → STOP) $\underset{\{a\}}{\|}$ ⊤). This computation corresponds to the execution of the left branch of the choice (i.e., `P`) and thus only event `b` occurs. Each rewriting step is labeled with the applied rule, and the example should be read top-down.

3 Tracing computations

In this section we define the notion of trace. Firstly, we introduce some notation that will be used along the paper. A trace is usually formed by the sequence of expressions that appear in the control of the semantics. These expressions are conveniently connected to form a graph. However, several program analysis techniques such as program slicing handle the source positions of programs, and thus, this notion of trace is insufficient for them. Therefore, we want our traces to store the source position of each expression so that the trace can be used to know

what portions of the source code have been executed and in what order. The inclusion of source positions in the trace implies an additional level of complexity in the semantics, but the benefits of providing our traces with this additional information are clear and, for some applications, essential. Therefore, we use labels (that we call *specification positions*) to identify each subexpression in a specification. Formally,

Definition 1. *(Position, specification position) Given a specification $\mathcal{S}$, the* positions *in $\mathcal{S}$ are represented by a sequence of natural numbers, where Λ denotes the empty sequence (i.e., the root position). They are used to address subexpressions of an expression viewed as a tree:*

$$
\begin{array}{ll}
Proc|_{\Lambda} = Proc & \forall\ Proc \in \mathcal{P} \\
(Proc\ op)|_{1.w} = Proc|_{w} & \forall\ op \in \{\backslash, [\![]\!]\} \\
(Proc_1\ op\ Proc_2)|_{1.w} = Proc_1|_{w} & \forall\ op \in \{\rightarrow, \sqcap, \Box, |||, ||, ;\} \\
(Proc_1\ op\ Proc_2)|_{2.w} = Proc_2|_{w} & \forall\ op \in \{\rightarrow, \sqcap, \Box, |||, ||, ;\}
\end{array}
$$

A specification position *is a pair (P, w) that addresses the subexpression $Proc|_w$ in the right-hand side of the process definition $P = Proc \in \mathcal{S}$. We often use $\mathcal{P}os(Proc)$ to refer to (P, w). In order to also identify names of processes, we use $(P, \_)$ as the specification position of P.*

We often use indistinguishably an expression and its associated specification position, when it is clear from the context. In order to introduce the formal definition of trace, we need first to define the concept of *control-flow*.

Definition 2. *(Control-flow) Given a CSP specification $\mathcal{S}$ and two specification positions n, n' in $\mathcal{S}$, we say that the control of n can pass to n' iff*

- $n = P \wedge n' = \mathit{first}((P, \Lambda))$ with $P = rhs(P) \in \mathcal{S}$
- $n \in \{\sqcap, \Box, |||, ||\} \wedge n' \in \{\mathit{first}(n.1), \mathit{first}(n.2)\}$
- $n \in \{\rightarrow, ;\} \wedge n' = \mathit{first}(n.2)$
- $n = n'.1 \wedge n' = \rightarrow$
- $n \in \mathit{last}(n'.1) \wedge n' = \ ;$
- $n \in \{\backslash, [\![]\!]\} \wedge n' = \mathit{first}(n.1)$

where $\mathit{first}(n)$ is the specification position of the subprocess denoted by n which must be executed first:

$$
\mathit{first}(n) = \begin{cases} n.1 & \text{if } n = \ \rightarrow \\ \mathit{first}(n.1) & \text{if } n = \ ; \\ n & \text{otherwise} \end{cases}
$$

and where $last(n)$ is the set of all possible termination points of the subprocess denoted by n:

$$last(n) = \begin{cases} \{n\} & \text{if } n = \texttt{SKIP} \\ \emptyset & \text{if } n = \texttt{STOP} \vee \\ & (n \in \{|||, ||\} \wedge (last(n.1) = \emptyset \vee last(n.2) = \emptyset)) \\ last(n.1) \cup last(n.2) & \text{if } n \in \{\sqcap, \Box\} \vee \\ & (n \in \{|||, ||\} \wedge last(n.1) \neq \emptyset \wedge last(n.2) \neq \emptyset) \\ last(n.2) & \text{if } n \in \{\rightarrow, ;\} \\ last(n.1) & \text{if } n \in \{\backslash, [\![]\!]\} \\ last((P, \Lambda)) & \text{if } n = P \end{cases}$$

Definition 3. *(Rewriting Step, Derivation) Given a CSP expression e, a* rewriting step *for e ($e \overset{\Theta}{\rightsquigarrow} e'$) is the application of a rule of the semantics of Fig. 1: $\frac{\Theta}{e \longrightarrow e'}$ where Θ is a (possibly empty) set of rewriting steps. Given a CSP expression e_0, we say that the sequence $e_0 \overset{\Theta_0}{\rightsquigarrow} \ldots \overset{\Theta_n}{\rightsquigarrow} e_{n+1}$, $n \geq 0$, is a* derivation *of e_0 iff $\forall\ i, 0 \leq i \leq n, e_i \overset{\Theta_i}{\rightsquigarrow} e_{i+1}$ is a rewriting step. We say that the derivation is* complete *iff there is no possible rewriting step for e_{n+1}. We say that the derivation has* successfully finished *iff e_{n+1} is $\top$.*

Function *last* can be used to determine the last specification position in a derivation. However, this function computes all possible final specification positions, and a derivation only reaches (non-deterministically) a set of them. Therefore, we will use in the following a modified version of *last* called *last′* whose behavior is exactly the same as *last* except in the case of choices. For each rewriting step $e \sqcap e' \overset{\Theta}{\rightsquigarrow} e$ or $e \Box e' \overset{\Theta_*}{\rightsquigarrow} e$, $last'(e \sqcap e') = last'(e \Box e') = e$.

Definition 4. *(Trace) Given a CSP specification $\mathcal{S}$, and a derivation $\mathcal{D}$ in $\mathcal{S}$, a* trace *of $\mathcal{D}$ is a graph $\mathcal{G} = (N, E_c, E_s)$ where N is a set of nodes labeled with specification positions and edges are divided into two groups:*

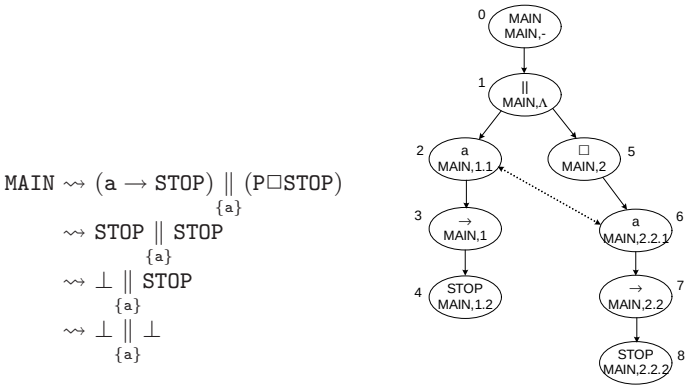
- control-flow edges *(E_c) are a set of one-way edges (denoted with $\mapsto$) representing the control-flow between two nodes, and*
- synchronization edges *(E_s) are a set of two-way edges (denoted with $\leftrightarrow$) representing the synchronization of two (event) nodes;*

and for each $e \overset{\Theta}{\rightsquigarrow} e' \in \mathcal{D}$ and for all rewriting steps in Θ we have that:

- *if e is a prefixing ($a \rightarrow P$) or a sequential composition ($Q; P$) and $\mathcal{P}os(P) \in N$, then E_c contains an edge $\mathcal{P}os(a) \mapsto \mathcal{P}os(\rightarrow)$ or $\mathcal{P}os(last'(Q)) \mapsto \mathcal{P}os(;)$ respectively,*
- *if the control can pass from e to e'' where $e'' \overset{\Theta'}{\rightsquigarrow} e''' \in \Theta$, then E_c contains an edge $\mathcal{P}os(e) \mapsto \mathcal{P}os(first(e''))$,*
- *if the control can pass from e to e', then E_c contains an edge $\mathcal{P}os(e) \mapsto \mathcal{P}os(first(e'))$, and*

– *E_s contains a synchronization edge $a \leftrightarrow a'$ for each synchronization occurring in the rewriting step where a and a' are the synchronized events.*

The only nodes in N are the nodes induced by E_c and E_s.



(a) Derivation (b) Trace

Fig. 3. Derivation and trace associated to the specification of Example 2

Example 2. Consider the specification of Example 1 where expressions are labeled with its associated specification positions (in grey color) so that labels are unique.

$$\texttt{MAIN}_{(\texttt{MAIN},\text{-})} = (\texttt{a}_{(\texttt{MAIN},1.1)} \rightarrow_{(\texttt{MAIN},1)} \texttt{STOP}_{(\texttt{MAIN},1.2)}) \underset{\{\texttt{a}\}}{\|}_{(\texttt{MAIN},\Lambda)} (\texttt{P}_{(\texttt{MAIN},2.1)} \,\square_{(\texttt{MAIN},2)}\, (\texttt{a}_{(\texttt{MAIN},2.2.1)} \rightarrow_{(\texttt{MAIN},2.2)} \texttt{STOP}_{(\texttt{MAIN},2.2.2)}))$$

$$\texttt{P}_{(\texttt{P},\text{-})} = \texttt{b}_{(P,1)} \rightarrow_{(P,\Lambda)} \texttt{SKIP}_{(P,2)}$$

We show in Fig. 3(a) one possible derivation (ignoring subderivations) of this specification. Its associated trace is shown in Fig. 3(b). For the time being, the reader can ignore the numbering of the nodes; they will be explained in Section 4. In the example, we see that the trace is a connected and directed graph. Apart from the control-flow edges, there is one synchronization edge between nodes $(\texttt{MAIN}, 2.2.1)$ and $(\texttt{MAIN}, 1.1)$ representing the synchronization of event a. We refer the reader to Appendix B where an example with loops is discussed.

4 Instrumenting the semantics for tracing

This section introduces an instrumented operational semantics of CSP which generates as a side-effect the trace associated to the computation performed with the semantics. The instrumented semantics is shown in Fig. 4, where we assume

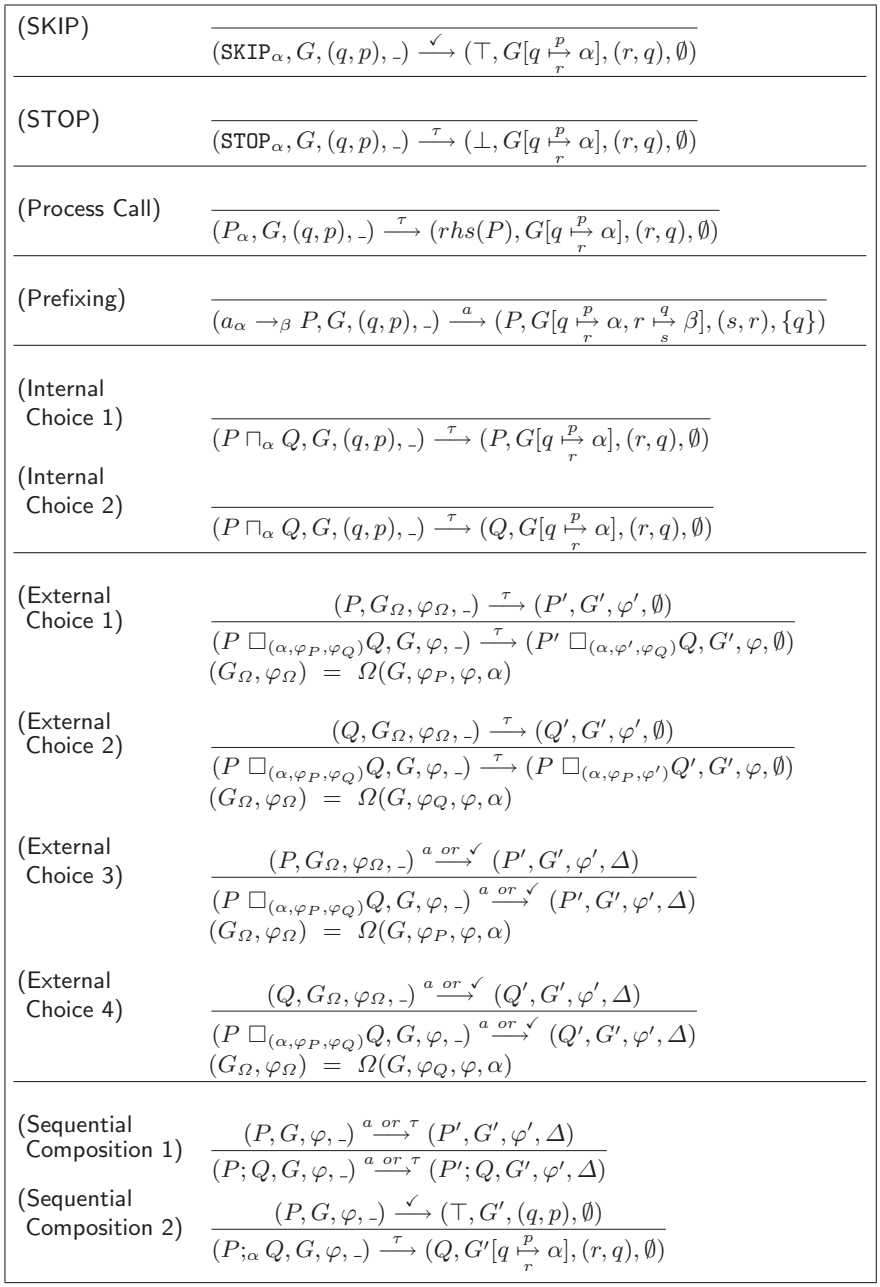


Fig. 4. An instrumented operational semantics for CSP that generates the trace

that every expression in the program has been labeled with its specification position (denoted by a subscript, e.g., P_α). In this semantics, a *state* is a tuple

(Synchronized Parallelism 1)
$$\frac{(P, G_\Omega, \varphi_\Omega, \_) \xrightarrow{a \text{ or } \{\tau \text{ or } \checkmark\}} (P', G', \varphi', \Delta)}{(P \underset{X}{\|}_{(\alpha,\varphi_P,\varphi_Q)} Q, G, \varphi, \_) \xrightarrow{a \text{ or } \tau} (P' \underset{X}{\|}_{(\alpha,\varphi',\varphi_Q)} Q, G', \varphi, \Delta)}$$
$$a \notin X \ \wedge \ (G_\Omega, \varphi_\Omega) \ = \ \Omega(G, \varphi_P, \varphi, \alpha)$$

(Synchronized Parallelism 2)
$$\frac{(Q, G_\Omega, \varphi_\Omega, \_) \xrightarrow{a \text{ or } \{\tau \text{ or } \checkmark\}} (Q', G', \varphi', \Delta)}{(P \underset{X}{\|}_{(\alpha,\varphi_P,\varphi_Q)} Q, G, \varphi, \_) \xrightarrow{a \text{ or } \tau} (P \underset{X}{\|}_{(\alpha,\varphi_P,\varphi')} Q', G', \varphi, \Delta)}$$
$$a \notin X \ \wedge \ (G_\Omega, \varphi_\Omega) \ = \ \Omega(G, \varphi_Q, \varphi, \alpha)$$

(Synchronized Parallelism 3)
$$\frac{RewritingStep_P \qquad RewritingStep_Q}{(P \underset{X}{\|}_{(\alpha,\varphi_P,\varphi_Q)} Q, G, \varphi, \_) \xrightarrow{a} (P' \underset{X}{\|}_{(\alpha,\varphi'_P,\varphi'_Q)} Q', G', \varphi, \Delta_P \cup \Delta_Q)}$$
$$a \in X \ \wedge \ G' = G_P \cup G_Q \cup \{s_P \overset{a}{\leftrightarrow} s_Q \mid s_P \in \Delta_P \wedge s_Q \in \Delta_Q\} \wedge$$
$$(G_{P_\Omega}, \varphi_{P_\Omega}) \ = \ \Omega(G, \varphi_P, \varphi, \alpha) \ \wedge \ (G_{Q_\Omega}, \varphi_{Q_\Omega}) \ = \ \Omega(G, \varphi_Q, \varphi, \alpha)$$
$$\wedge \ RewritingStep_P = (P, G_{P_\Omega}, \varphi_{P_\Omega}, \_) \xrightarrow{a} (P', G_P, \varphi'_P, \Delta_P)$$
$$\wedge \ RewritingStep_Q = (Q, G_{Q_\Omega}, \varphi_{Q_\Omega}, \_) \xrightarrow{a} (Q', G_Q, \varphi'_Q, \Delta_Q)$$

(Synchronized Parallelism 4)
$$\frac{}{(\top \underset{X}{\|}_{(\alpha,(q_P,p_P),(q_Q,p_Q))} \top, G, \varphi, \_) \xrightarrow{\checkmark} (\top, G[p_P \underset{r}{\mapsto}, p_Q \underset{r}{\mapsto}], (r, p_P), \emptyset)}$$

(Hiding 1)
$$\frac{(P, G_\Sigma, \varphi_\Sigma, \_) \xrightarrow{a} (P', G', \varphi', \_)}{(P \backslash_\alpha B, G, \varphi, \_) \xrightarrow{\tau} (P' \backslash_\bullet B, G', \varphi', \emptyset)} \quad a \in B$$

(Hiding 2)
$$\frac{(P, G_\Sigma, \varphi_\Sigma, \_) \xrightarrow{a \text{ or } \tau} (P', G', \varphi', \Delta)}{(P \backslash_\alpha B, G, \varphi, \_) \xrightarrow{a \text{ or } \tau} (P' \backslash_\bullet B, G', \varphi', \Delta)} \quad a \notin B$$

(Hiding 3)
$$\frac{(P, G_\Sigma, \varphi_\Sigma, \_) \xrightarrow{\checkmark} (\top, G', \varphi', \emptyset)}{(P \backslash_\alpha B, G, \varphi, \_) \xrightarrow{\checkmark} (\top, G', \varphi', \emptyset)}$$
where $(G_\Sigma, \varphi_\Sigma) \ = \ \Sigma(G, \alpha, \varphi)$

(Renaming 1)
$$\frac{(P, G_\Sigma, \varphi_\Sigma, \_) \xrightarrow{a} (P', G', \varphi', \Delta)}{(P[\![R]\!]_\alpha, G, \varphi, \_) \xrightarrow{b} (P'[\![R]\!]_\bullet, G', \varphi', \Delta)} \quad (a, b) \in R$$

(Renaming 2)
$$\frac{(P, G_\Sigma, \varphi_\Sigma, \_) \xrightarrow{a \text{ or } \tau} (P', G', \varphi', \Delta)}{(P[\![R]\!]_\alpha, G, \varphi, \_) \xrightarrow{a \text{ or } \tau} (P'[\![R]\!]_\bullet, G', \varphi', \Delta)} \quad (a, \_) \notin R$$

(Renaming 3)
$$\frac{(P, G_\Sigma, \varphi_\Sigma, \_) \xrightarrow{\checkmark} (\top, G', \varphi', \emptyset)}{(P[\![R]\!]_\alpha, G, \varphi, \_) \xrightarrow{\checkmark} (\top, G', \varphi', \emptyset)}$$
where $(G_\Sigma, \varphi_\Sigma) \ = \ \Sigma(G, \alpha, \varphi)$

Fig. 4. An instrumented operational semantics for CSP that generates the trace (cont.)

$(e, G, (q, p), \Delta)$, where e is the expression to be evaluated (the *control*), G is a directed graph (i.e., the trace built so far), (q, p) are references to the current node and its parent in G, and Δ is a set of references to nodes that must be synchronized. Concretely, q is a reference to store the specification position of

the current expression e in the control, and p denotes the parent of q. q will be often a fresh[1] reference generated to add new nodes to G. The basic idea of the graph construction is to record the current control with the current reference in every step by connecting it to its parent whose reference is p. We use the notation $G[q \overset{p}{\underset{r}{\mapsto}} \alpha]$ to introduce a node in G. For instance, if we are adding a node to G this new node has reference q, is labeled with specification position α, its parent is p, and its successor is r. It is assumed that the reference of the root node of the trace (e.g., $(\texttt{MAIN}, \Lambda)$) is called *Root*.

An explanation for each rule of the semantics follows:

(SKIP and STOP) Whenever one of these rules is applied, the subcomputation finishes because $\top$ (for rule `SKIP`) and $\bot$ (for rule `STOP`) are put in the control, and these special symbols have no associated rule. A node with the `SKIP` (respectively `STOP`) specification position is added to the graph.

(Process Call) The called process is unfolded, node q is added to the graph with specification position α, parent p and successor r (a fresh reference). In the new state, q becomes the parent reference and the fresh reference r represents the current reference. The new expression in the control is $rhs(P)$. The set Δ of events to be synchronized is initialized to $\emptyset$.

(Prefixing) This rule adds nodes q (the prefix) and r (the prefixing operator) to the graph. Node q is the parent of r. In the new state, r becomes the parent reference and the fresh reference s represents the current reference. The new control is P. The set Δ is initialized to $\{q\}$ to indicate that event a has occurred and it must be synchronized if required by other rule.

(Internal Choice 1 and 2) The choice operator is added to the graph, and the (non-deterministically) selected branch is put into the control with the fresh reference r as the child of the choice operator.

(External Choice 1, 2, 3 and 4) External choices can develop both branches while τ events happen, until an event in $\Sigma \cup \{\checkmark\}$ occurs. This means that the semantics can add nodes to both branches of the trace alternatively, and thus, it needs to store the next reference to use in every branch of the choice. This is done by labeling choice operators with a tuple of the form $(\alpha, \varphi, \varphi')$ where α is the specification position of the choice operator; and φ and φ' are respectively the references to be used in the left and right branches of the choice, and they are initialized to $(\_,\_)$. Therefore, these rules can be fired several times to evolve the branches of the choice, but the choice operator must be introduced into the graph only once (i.e., the first time). For this purpose, function Ω is used:

$$\Omega(G, \varphi, (q, p), \alpha) \;=\; \begin{cases} (G[q \overset{p}{\underset{r}{\mapsto}} \alpha], (r, q)) & \text{if } \varphi = (\_, \_) \\ (G, \varphi) & \text{otherwise} \end{cases}$$

[1] We assume that fresh references are generated taking the next node identifier not used into account (i.e., fresh references already generated but not used are employed before producing new references).

This function checks whether this is the first time that the branch is evaluated (this only happens when the references of this branch are empty, i.e., $\varphi = (\_, \_)$). In this case, the choice operator is added to G.

(Sequential Composition 1 and 2) Sequential Composition 1 is used to evolve process P until it is finished. P is evolved to P' which is put into the control. When P finishes (it becomes $\top$), $\checkmark$ happens. Then, Sequential Composition 2 is used and Q is put into the control. The sequential composition operator ; is added to the graph with parent p that is the reference to the last node added in the subderivation associated to P.

(Synchronized Parallelism 1 and 2) In a parallelism, both parallel processes can be intertwiningly executed until a synchronized event is found. Therefore, nodes from both processes can be added interweaved to the graph. Hence, parallelism operators are labelled with a tuple of the form $(\alpha, \varphi, \varphi')$ as it happens with external choices.

These rules develop the branches of the parallelism until they are finished or until they must synchronize. They use function Ω to introduce the parallelism into the graph the first time it is executed and only if it has not been introduced in a previous rewriting step.

(Synchronized Parallelism 3) This rule is used to synchronize the parallel processes. In this case, both branches must perform a rewriting step with the same visible (and synchronized) event. Each branch has a non-empty set of events Δ to be synchronized (note that this is a set because many parallelisms could be nested). Then, all references in the sets Δ_P and Δ_Q are mutually linked with a synchronization edge.

(Synchronized Parallelism 4) It is used when none of the parallel processes can proceed because they already finished. In this case, the control becomes $\top$ indicating the successful termination of the synchronized parallelism. In the new state, the new (fresh) reference is r. Therefore, this rule also adds to the graph two arcs from the last references of each branch (p_P and p_Q) to r.

(Hiding 1, 2 and 3) Hiding 1 is used to hide an event in P that belongs to the hiding set B. In this case, the event is hidden with τ. Because the event is hidden, it should not be synchronized by other rules; therefore, the set Δ is initialized to $\emptyset$. If the event does not belong to B then Hiding 2 is used, and thus, it remains observable from outside. In both cases, the specification position of the hiding operator is replaced by $\bullet$ (in the next state of the semantics, not in the specification) meaning that it has been already evaluated. This is used to ensure that the hiding operator is only added to the graph once. For this purpose, function Σ is used:

$$\Sigma(G, \alpha, (q, p)) \;=\; \begin{cases} (G[q \overset{p}{\underset{r}{\mapsto}} \alpha], (r, q)) & \text{if } \alpha \neq \bullet \\ (G, (q, p)) & \text{otherwise} \end{cases}$$

Function Σ checks that the specification position of the hiding operator is not $\bullet$. In this case, it is the first time that it is evaluated and thus it is added to the graph. Hiding 3 is used to finish the process by placing $\top$ in the control and performing $\checkmark$.

(Renaming 1, 2 and 3) It is completely analogous to the previous case, but in this case, the event is not hidden, but replaced by another event in mapping R. Note in Renaming 1 that, contrarily to Hiding 1, the set Δ is passed down from the top rewriting step. This is done because an event a happened that has been added to Δ and must be synchronized if required by other rule. Due to the renaming, other rules see a as b, so, if they try to synchronize b, they will use the reference of a introduced in Δ.

We illustrate this semantics with a simple example.

Example 3. Consider again the specification in Example 2. The execution of the instrumented semantics with the initial state $(\texttt{MAIN}, \emptyset, (0, \mathit{Root}), \emptyset)$ gets the computation of Fig. 5. Here, for clarity, each computation step is labeled with the applied rule; in each state, G denotes the current graph. This computation corresponds to the execution of the right branch of the choice (i.e., $\texttt{a} \rightarrow \texttt{STOP}$). The final state is $(\bot \underset{\{\texttt{a}\}}{\parallel}_{((\texttt{MAIN},\Lambda),(9,4),(10,8))} \bot, G', (1,0), \emptyset)$. The final trace G' computed for this execution is depicted in Fig. 3(b) where we can see that nodes are numbered with the references generated by the instrumented semantics.

5 Correctness

In this section we prove the correctness of the semantics by showing that (i) the computations performed by the instrumented semantics are equivalent to the computations performed by the standard semantics; and (ii) the graph produced by the instrumented semantics is the trace of the derivation.

Theorem 1 (Conservativeness). *Let $\mathcal{S}$ be a CSP specification, e an expression in $\mathcal{S}$, and $\mathcal{D}$ and $\mathcal{D}'$ the derivations of e performed with the semantics of Fig. 1 and Fig. 4, respectively. Then, the sequence of rules applied in $\mathcal{D}$ and $\mathcal{D}'$ is exactly the same except that $\mathcal{D}'$ performs one rewriting step more than $\mathcal{D}$ for each (sub)computation that finishes with* `STOP`*.*

This theorem shows that the computations performed with the instrumented semantics are all and only the computations performed with the standard semantics. The only difference between them from an operational point of view is that the instrumented semantics needs to perform one step when a `STOP` is evaluated (to add its specification position to the trace) and then finishes, while the standard semantics finishes without performing any step. The proof of the theorem can be found in Appendix C.

The following theorem states the correctness of the instrumented semantics by ensuring that the graph produced is a trace of the computation.

Theorem 2 (Correctness). *Let $\mathcal{S}$ be a CSP specification, $\mathcal{D}$ a derivation of $\mathcal{S}$ performed with the semantics of Fig. 4, and $\mathcal{G}$ the graph produced by $\mathcal{D}$. Then, $\mathcal{G}$ is the trace associated to $\mathcal{D}$.*

The proof of this theorem can be found in Appendix C.

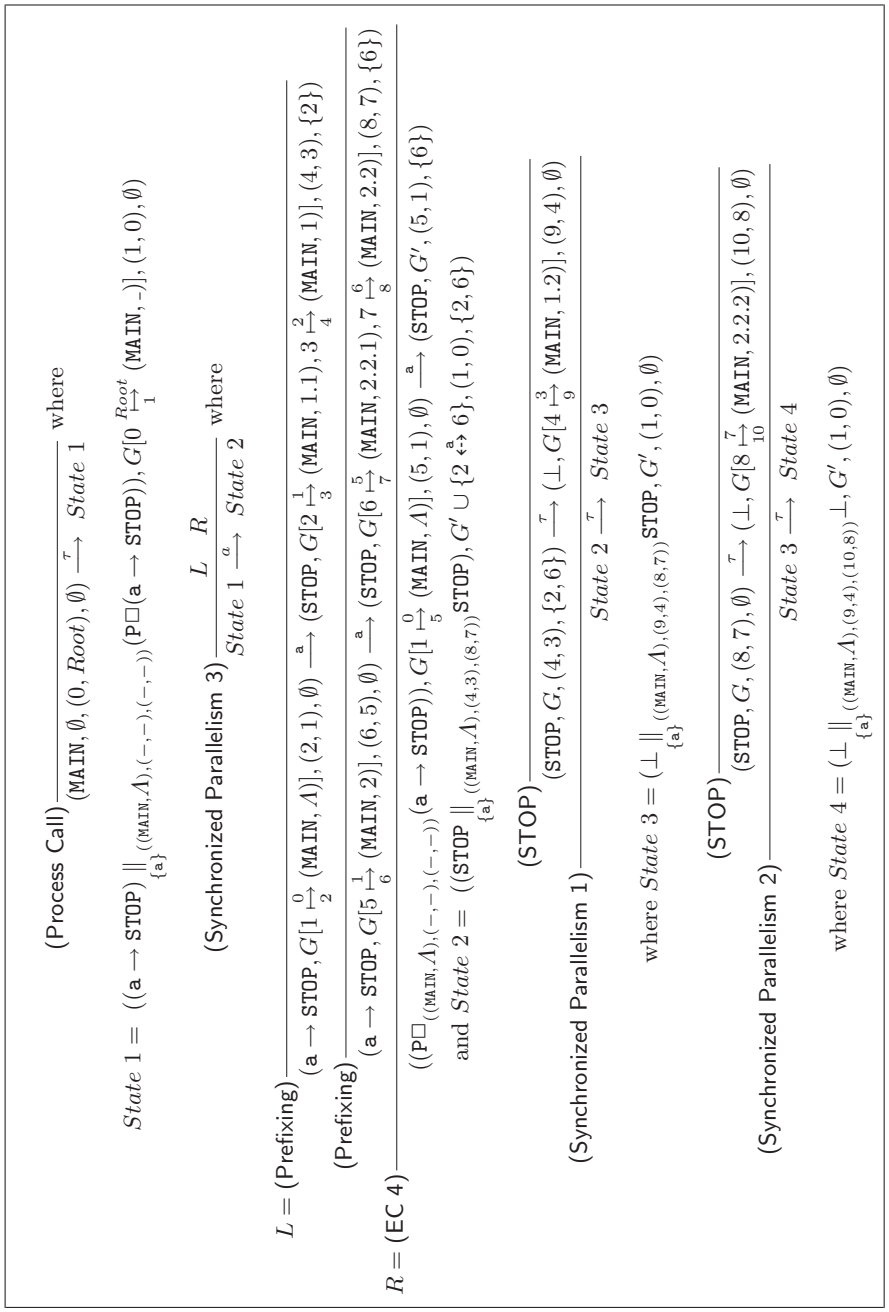


Fig. 5. An example of computation with the semantics in Fig. 4

6 Conclusions

This work introduces the first semantics of CSP instrumented for tracing. The execution of the instrumented semantics produces a graph as a side effect which is the trace of the computation. This trace is produced by the semantics step by step, and thus, it can be also used to produce a trace of an infinite computation until has been stopped. The generated trace can be useful not only for tracing computations but for debugging and program comprehension. This is due to the fact that our generated trace also includes the specification positions associated to the expressions appearing in the trace. Therefore, traces could be used to analyse what parts of the program are executed (and in what order) in a particular computation. Also, this information allows a traces viewer tool to highlight the parts of the code that are executed in each step.

This semantics is the first tracing semantics for CSP. Therefore, it is an interesting result because it can serve as a reference mark to prove properties such as completeness of static analyses which are based on traces.

References

1. O. Chitil. A Semantics for Tracing. In *13th Intl Workshop on Implementation of Functional Languages (IFL 2001)*, pp. 249-254. Ericsson CSL, 2001.
2. O. Chitil, C. Runciman, M. Wallace. Transforming Haskell for Tracing. In *Int'l Workshop on the Impl. of Functional Languages*, LNCS 2670, pp. 165–181, 2003.
3. Hoare, C. A. R. *Communicating sequential processes.* Communications ACM, 26(1):100–106, 1983.
4. Kavi, K. M., F. T. Sheldon, and B. Shirazi. Reliability analysis of CSP specifications using Petri nets and Markov processes. In *Proceedings 28th Annual Hawaii International Conference on System Sciences. Software Technology*, vol. 2, pp. 516–524, Wailea, HI, 1995.
5. Ladkin, P. and B. Simons. Static deadlock analysis for csp-type communications. *Responsive Computer Systems (Chapter 5)*, Kluwer Academic Publishers, 1995.
6. Leuschel, M. and M. Butler. ProB: an automated analysis toolset for the B method. *Journal of Software Tools for Technology Transfer*, vol. 10(2), pp. 185–203, 2008.
7. Leuschel, M., M. Llorens, J. Oliver, J. Silva, and S. Tamarit, Static slicing of CSP specifications. In *Proceedings of the 18th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'08)*, pp. 141–150, 2008.
8. Leuschel, M., M. Llorens, J. Oliver, J. Silva, and S. Tamarit, SOC: a slicer for CSP specifications. In *Proceedings of the 2009 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM'09)*, pp. 165–168, Savannah, GA, USA, 2009.
9. Leuschel, M., M. Llorens, J. Oliver, J. Silva, and S. Tamarit, The MEB and CEB Static Analysis for CSP Specifications. In *Post-proceedings of the 18th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'08)*, Revised Selected Papers, LNCS 5438, Springer-Verlag, pp. 103–118, 2009.
10. Roscoe, A.W., The Theory and Practice of Concurrency, Prentice-Hall, 2005.
11. M. D. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, vol. 10(4), pp. 352–357, 1984.

A The syntax of CSP

For those readers not familiarized with CSP, this appendix recalls CSP's syntax.

S	$::= D_1 \dots D_m$	(entire specification)	*Domains* $P, Q, R \dots \in \mathcal{P}$ (processes) $a, b, c \dots \in \Sigma$ (events) $u, v, w \dots \in \mathcal{V}$ (variables)
D	$::= P = Proc$	(process definition)	
$Proc$	$::= Q$	(process call)	
	$\mid\ x \rightarrow Proc$	(prefixing)	
	$\mid\ c?u \rightarrow Proc$	(input)	
	$\mid\ c!u \rightarrow Proc$	(output)	
	$\mid\ Proc_1 \sqcap Proc_2$	(internal choice)	
	$\mid\ Proc_1 \,\Box\, Proc_2$	(external choice)	
	$\mid\ Proc_1 \,\|\|_{\{\overline{x_n}\}}\, Proc_2$	(synchronized parallelism)	where $\overline{x_n} = x_1, \dots, x_n$ and $x_i \in \Sigma \cup \mathcal{V}$
	$\mid\ Proc_1 \,\|\|\|\, Proc_2$	(interleaving)	
	$\mid\ Proc_1 \,;\, Proc_2$	(sequential composition)	
	$\mid\ Proc \backslash \{\overline{x_n}\}$	(hiding)	
	$\mid\ Proc[\![f]\!]$	(renaming)	where $f : \Sigma \rightarrow \Sigma$
	$\mid\ SKIP$	(skip)	
	$\mid\ STOP$	(stop)	

Fig. 6. Syntax of CSP specifications

Figure 6 summarizes the syntax constructions used in CSP specifications. A specification is a finite collection of definitions. The left-hand side of each definition is the name of a different process, which is defined in the right-hand side by means of an expression that can be a call to another process or a combination of the following operators:

Prefixing. It specifies that event x must happen before expression $Proc$.

Input. It is used to receive a message from another process. Message u is received through channel c; then process $Proc$ is executed.

Output. It is analogous to the input, but this is used to send messages. Message u is sent through channel c; then process $Proc$ is executed.

Internal choice. The system chooses (e.g., non-deterministically) to execute one of the two expressions.

External choice. It is identic to internal choice but the choice comes from outside the system (e.g., the user).

Synchronized parallelism. Both expressions are executed in parallel with a set of synchronized events. In absence of synchronizations both expressions can execute in any order. Whenever a synchronized event $x_i, 1 \leq i \leq n$,

happens in one of the expressions it must also happen in the other at the same time. Whenever the set of synchronized events is not specified, it is assumed that expressions are synchronized in all common events.

Interleaving. Both expressions are executed in parallel and independently. Therefore, it is operationally equivalent to a parallelism with an empty synchronization events set.

Sequential composition. It specifies a sequence of two processes. When the first (successfully) finishes, the second starts.

Hiding. Process *Proc* is executed with a set of hidden events $\{\overline{x_n}\}$. Hidden events are not observable from outside the process, and thus, they cannot synchronize with other processes.

Renaming. Process *Proc* is executed with a set of renamed events specified with the mapping f. An event a renamed as b behaves internally as a but it is observable as b from outside the process.

Skip. It finishes the current process. It allows us to continue the next sequential process.

Stop. It finishes the current process; but it does not allow the next sequential process to continue.

B Tracing a specification with loops

In this appendix we show another example where a non-terminating process appears which execution finishes with a deadlock.

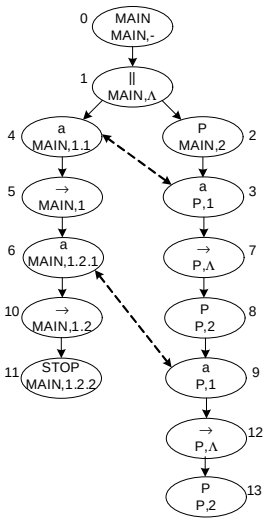


Fig. 7. Trace of the program in Example 4

$$\text{(Process Call)}\frac{}{(\texttt{MAIN}, \emptyset, (0, Root), \emptyset) \xrightarrow{\tau} State\ 1} \text{ where}$$

$$State\ 1 = ((\texttt{a} \to \texttt{a} \to \texttt{STOP} \underset{\{\texttt{a}\}}{\parallel}_{((\texttt{MAIN}, \Lambda), (-,-), (-,-), \emptyset)} \texttt{P}), G[0 \overset{Root}{\underset{1}{\mapsto}} (\texttt{MAIN}, \_)], (1,0), \emptyset)$$

$$\text{(Syncronized Parallelism 2)}\frac{\text{(Process Call)}\dfrac{}{(\texttt{P}, G[1 \overset{0}{\underset{2}{\mapsto}} (\texttt{MAIN}, \Lambda)], (2,1), \emptyset) \xrightarrow{\tau} (\texttt{a} \to \texttt{P}, G[2 \overset{1}{\underset{3}{\mapsto}} (\texttt{MAIN}, 2)], (3,2), \emptyset)}}{State\ 1 \xrightarrow{\tau} State\ 2}$$

$$\text{where } State\ 2 = ((\texttt{a} \to \texttt{a} \to \texttt{STOP} \underset{\{\texttt{a}\}}{\parallel}_{((\texttt{MAIN}, \Lambda), (-,-), (3,2), \emptyset)} \texttt{a} \to \texttt{P}), G', (1,0), \emptyset)$$

$$\text{(Synchronized Parallelism 3)}\frac{L \qquad R}{State\ 2 \xrightarrow{\texttt{a}} State\ 3} \text{ where}$$

$$L = \text{(Prefixing)}\frac{}{(\texttt{a} \to \texttt{a} \to \texttt{STOP}, G[1 \overset{0}{\underset{4}{\mapsto}} (\texttt{MAIN}, \Lambda)], (4,1), \emptyset) \xrightarrow{\texttt{a}} (\texttt{a} \to \texttt{STOP}, G[4 \overset{1}{\underset{5}{\mapsto}} (\texttt{MAIN}, 1.1), 5 \overset{4}{\underset{6}{\mapsto}} (\texttt{MAIN}, 1)], (6,5), \{4\})}$$

$$R = \text{(Prefixing)}\frac{}{(\texttt{a} \to \texttt{P}, G, (3,2), \emptyset) \xrightarrow{\texttt{a}} (\texttt{P}, G[3 \overset{2}{\underset{7}{\mapsto}} (\texttt{P}, 1), 7 \overset{3}{\underset{8}{\mapsto}} (\texttt{P}, \Lambda)], (8,7), \{3\})}$$

$$\text{and } State\ 3 = (\texttt{a} \to \texttt{STOP} \underset{\{\texttt{a}\}}{\parallel}_{((\texttt{MAIN}, \Lambda), (6,5), (8,7))} \texttt{P}, G' \cup \{4 \overset{\texttt{a}}{\leftrightarrow} 3\}, (1,0), \{4,3\})$$

$$\text{(Syncronized Parallelism 2)}\frac{\text{(Process Call)}\dfrac{}{(\texttt{P}, G, (8,7), \{4,3\}) \xrightarrow{\tau} ((\texttt{a} \to \texttt{P}), G[8 \overset{7}{\underset{9}{\mapsto}} (\texttt{P}, 2)], (9,8), \emptyset)}}{State\ 3 \xrightarrow{\tau} State\ 4}$$

$$\text{where } State\ 4 = ((\texttt{a} \to \texttt{STOP}) \underset{\{\texttt{a}\}}{\parallel}_{((\texttt{MAIN}, \Lambda), (6,5), (9,8))} (\texttt{a} \to \texttt{P}), G', (1,0), \emptyset)$$

Fig. 8. Computation of Example 4 with the semantics in Fig. 4

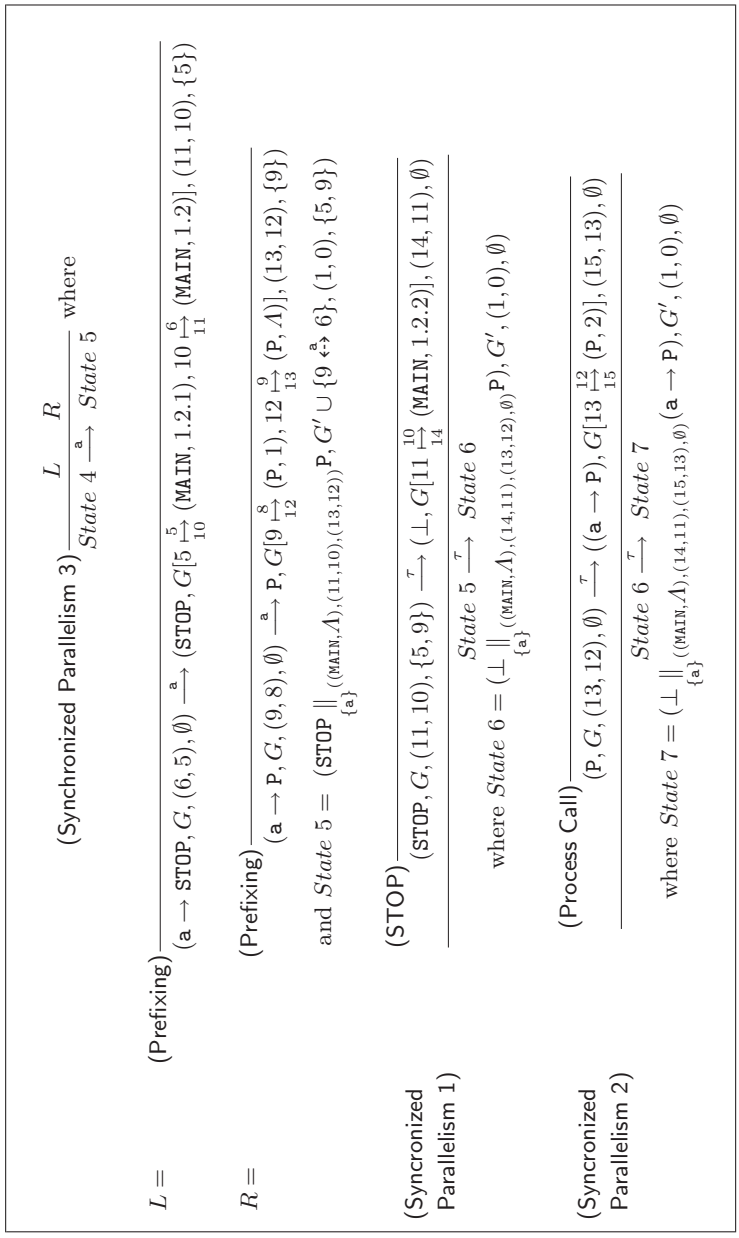


Fig. 8. Computation of Example 4 with the semantics in Fig. 4 (cont.)

Example 4. Consider the following CSP specification where each expression has been labeled with its associated specification position (in grey color).

$$\texttt{MAIN} = \texttt{a}_{(\texttt{MAIN},1.1)} \rightarrow_{(\texttt{MAIN},1)} \texttt{a}_{(\texttt{MAIN},1.2.1)} \rightarrow_{(\texttt{MAIN},1.2)} \texttt{STOP}_{(\texttt{MAIN},1.2.2)} \underset{\{\texttt{a}\}}{\parallel}_{(\texttt{MAIN},\Lambda)} \texttt{P}_{(\texttt{MAIN},2)}$$

$$\texttt{P} = \texttt{a}_{(\texttt{P},1)} \rightarrow_{(\texttt{P},\Lambda)} \texttt{P}_{(\texttt{P},2)}$$

We use the initial state $(\texttt{MAIN}, \emptyset, (0, \mathit{Root}), \emptyset)$ to execute the semantics and get the computation of Fig. 8. The final state is $(\bot \underset{\{\texttt{a}\}}{\parallel}_{((\texttt{MAIN},\Lambda),(14,11),(15,13))} (\texttt{a} \rightarrow \texttt{P}), G', (1,0), \emptyset)$. The final trace G' computed is the graph of Fig. 7.

C Proofs of technical results

In this appendix we present the proofs of the technical results of the paper.

Theorem 1 (Conservativeness). *Let $\mathcal{S}$ be a CSP specification, e an expression in $\mathcal{S}$, and $\mathcal{D}$ and $\mathcal{D}'$ the derivations of e performed with the semantics of Fig. 1 and Fig. 4, respectively. Then, the sequence of rules applied in $\mathcal{D}$ and $\mathcal{D}'$ is exactly the same except that $\mathcal{D}'$ performs one rewriting step more than $\mathcal{D}$ for each (sub)computation that finishes with* `STOP`*.*

Proof. Firstly, rule (STOP) of the instrumented semantics is the only rule that is not present in the standard semantics. When a `STOP` is reached in a derivation, the standard semantics stops the (sub)computation because no rule is applicable. In the instrumented semantics, when a `STOP` is reached in a derivation, the only rule applicable is (STOP) which performs τ and puts $\bot$ in the control. Then, the (sub)computation is stopped because no rule is applicable for $\bot$. Therefore, when the control in the derivation is `STOP`, the instrumented semantics performs one additional rewriting step with rule (STOP).

The claim follows from the fact that both semantics have exactly the same number of rules except for rule (STOP), and these rules have the same control in the four states of the rules. Therefore, all derivations in both semantics have exactly the same number of steps and they are composed of the same sequences of rewriting steps except for (sub)computations finishing with `STOP` that perform one rewriting step more (applying rule (STOP)). □

Now we are going to prove Theorem 2.

Theorem 2 (Correctness) *Let $\mathcal{S}$ be a CSP specification, $\mathcal{D}$ a derivation of $\mathcal{S}$ performed with the semantics of Fig. 4, and $\mathcal{G}$ the graph produced by $\mathcal{D}$. Then, $\mathcal{G}$ is the trace associated to $\mathcal{D}$.*

In order to prove the correctness of the semantics, the following lemmas are necessary.

Lemma 1. *Let $\mathcal{S}$ be a CSP specification, $\mathcal{D}$ a complete derivation of $\mathcal{S}$ performed with the semantics of Fig. 4, and $\mathcal{G}$ the graph produced by $\mathcal{D}$. Then, for each prefixing $(a \rightarrow P)$ in the control of the left state of a rewriting step in $\mathcal{D}$, we have that $\mathcal{P}os(a)$ and $\mathcal{P}os(\rightarrow)$ are nodes of $\mathcal{G}$ and $\mathcal{P}os(\rightarrow)$ is the successor of $\mathcal{P}os(a)$.*

Proof. If a prefixing $a \rightarrow P$ is in the control of the left state of a rewriting step, following the semantics of Fig. 4, only the rule (Prefixing) can be applied. By definition of this rule, (q, p) are the references for the current (q) and parent (p) nodes in $\mathcal{G}$. The rule (Prefixing) adds two new nodes to the graph: q and r. The node q is labeled with the specification position of event a and has parent p and successor r. The node r is labeled with the specification position of operator $\rightarrow$ and has parent q and successor s (a fresh reference). Therefore, we have that $\mathcal{P}os(a)$ and $\mathcal{P}os(\rightarrow)$ are nodes of $\mathcal{G}$ and $\mathcal{P}os(\rightarrow)$ is the successor of $\mathcal{P}os(a)$. □

Lemma 2. *Given a derivation $\mathcal{D}$, for each rewriting step $e \overset{\Theta}{\rightsquigarrow} e'$ in $\mathcal{D}$ with $e' \neq \top$ and $e' \neq \bot$, either $last'(e) = e'$ or $last'(e) = last'(e')$.*

Proof. We prove this lemma by induction on the length of Θ. In the base case, Θ is empty, and thus only the rules (Process Call), (Prefixing) and (Internal Choice 1 and 2) can be applied. In all cases, the lemma holds trivially by the definition of $last'$. We assume as the induction hypothesis that the lemma holds for a non-empty Θ_i with $i > 0$ rewriting steps; and prove that the lemma also holds for a Θ_{i+1} with $i + 1$ rewriting steps. We can assume that $\Theta_{i+1} = e \overset{\Theta_i}{\rightsquigarrow} e'$, thus, we have to prove that the lemma holds for any possible e and e'. The possible cases are the following:

(External Choice 1 and 2) This case is trivial because the specification position of e and e' are the same. Hence, $last'(e) = last'(e')$.

(External Choice 3 and 4) Both cases are similar. Thus, we only discuss (External Choice 3). In the case of (External Choice 3), $last'(P \,\Box\, Q) = last'(P)$. This rule puts P' in the control, and we know by the induction hypothesis that $last'(P) = last'(P')$ and, thus, the lemma holds.

(Sequential Composition 1) This case is analogous to (External Choice 1 and 2).

(Sequential Composition 2) $last'(P; Q) = last'(Q)$. Therefore the lemma holds trivially by the definition of $last'$.

(Synchronized Parallelism 1, 2 and 3) It is the same case as (External Choice 1).

(Hiding 1 and 2) This case is similar to (External Choice 3). Note that the loss of the specification position (it is $\bullet$ in e') is only used as a flag to add nodes to the graph; but it does not have any influence on the derivation.

(Renaming 1 and 2) It is completely analogous to the previous case.

□

Lemma 3. *Let $\mathcal{S}$ be a CSP specification, $\mathcal{D}$ a complete derivation of $\mathcal{S}$ performed with the semantics of Fig. 4, and $\mathcal{G}$ the graph produced by $\mathcal{D}$. Then, for each sequential composition $(P; Q)$ in the control of the left state of a rewriting step in $\mathcal{D}$, we have that $\mathcal{P}os(last'(P))$ and $\mathcal{P}os(;)$ are nodes of $\mathcal{G}$ and $\mathcal{P}os(;)$ is the successor of $\mathcal{P}os(last'(P))$ whenever P has successfully finished.*

Proof. If a sequential composition $(P; Q)$ is in the left state of the control of a rewriting step, following the semantics of Fig. 4, only the rules (Sequential Composition 1) and (Sequential Composition 2) can be applied. (Sequential Composition 1) is only used to evolve process P until it is finished. The application of this rule

is only possible with any event except ✓, remaining the sequential composition operator in the control. (Sequential Composition 2) can only be used when ✓ happens and thus $\top$ is in the control.

Therefore, when P has successfully finished evolving to $\top$ and with (q, p) as references, (Sequential Composition 2) is applied. This rule adds to the graph a new node q labeled with the specification position of ; that has parent p and successor r (a fresh reference). Therefore, we have that $\mathcal{P}os(;)$ is a node of $\mathcal{G}$ and $\mathcal{P}os(;)$ is the successor of p. Then, we have to prove that $\mathcal{P}os(last'(P))$ is a node of the graph added before P successfully finished with reference p.

We prove this claim by induction on the length of the derivation $P \overset{\Theta_*}{\rightsquigarrow} \top$. The base case happens when the last rewriting step of the derivation is done leaving $\top$ in the control. Only these rules can be used:

(SKIP) In this case, $q \overset{p}{\underset{r}{\mapsto}} \texttt{SKIP}$ is added to $\mathcal{G}$ and (r, q) are the new references. $last'(\texttt{SKIP}) = \texttt{SKIP}$. Therefore the claim follows.

(External Choice 3) Here, $last'(P \,\Box\, Q) = last'(P)$. This rule puts P' in the control which is $\top$ by the conditions of the lemma. Therefore, there must be a SKIP, which is $last'(P)$, at the top of Θ because we know that the derivation successfully finishes and thus Θ is finite.

(External Choice 4) It is analogous to the previous case, but here $last'(P \,\Box\, Q) = last'(Q)$.

(Synchronized Parallelism 4) $last'(P \mid\mid Q) = last'(P) \cup last'(Q)$. The last nodes of P and Q, p_P and p_Q respectively, are added to $\mathcal{G}$ and connected to the new reference r. Therefore the claim follows.

(Hiding 3) In this case, because $last'(P\backslash B) = last'(P)$ and P is put in the control of the left state in the rewriting step of the precondition which must be reduced to $\top$ performing ✓, the claim follows by the recursive application of one of these six rules.

(Renaming 3) It is completely analogous to the previous case.

The induction hypothesis states that for all rewriting step $e \overset{\Theta}{\rightsquigarrow} e', e' \neq \top$ in the derivation $Q \overset{\Theta_*}{\rightsquigarrow} \top$ where $P \overset{\Theta}{\rightsquigarrow} Q \in \mathcal{D}$, $last'(P)$ is put in the control of a further rewriting step of the derivation together with its reference.

Then, we prove that this also holds for the previous rewriting step $e_0 \overset{\Theta}{\rightsquigarrow} e$. Only rules that do not perform ✓ could be applied (because ✓ puts $\top$ in the control of the right state and now, we are not considering the final rewriting step).

(STOP) This rule could not be applied because it puts $\bot$ in the control. There is no rule for $\bot$ thus, if applied, P could not successfully finished.

(Process Call), (Prefixing) and (Internal Choice 1 and 2) In these rules e is put in the control of the final state together with its reference. We know that $last'(e_0) = last'(e)$ thus, the claim follows by the induction hypothesis.

(External Choice 1 and 2) Both rules keep the expression in the control and the same references, thus the claim follows by the induction hypothesis.

(External Choice 3 and 4) In this case, $last'(P \,\square\, Q) = last'(P)$. This rule puts P' in the control, and we know by Lemma 2 that $last'(P) = last'(P')$, thus the lemma holds by the induction hypothesis.

(Sequential Composition 1 and 2) We know that P successfully finished, thus (Sequential Composition 1) is applied a number of times before (Sequential Composition 2), that puts Q in the control. We know that $last'(P;Q) = last'(Q)$ thus, the claim holds by the induction hypothesis.

(Synchronized Parallelism 4) $last'(P||Q) = last'(P) \cup last'(Q)$. The last nodes of P and Q, p_P and p_Q respectively, are added to $\mathcal{G}$ and connected to the new reference r. Therefore the claim follows.

(Hiding 3) In this case, because $last'(P\backslash B) = last'(P)$ and P is put in the control of the condition state which must be reduced to $\top$ performing $\checkmark$, the claim follows by the recursive application of one of these rules.

(Renaming 3) It is completely analogous to the previous case.

□

In the following lemma, we need to extend the notion of rewriting step by including the graph references. Therefore, we will use *extended* rewriting step denoted with $(e, \varphi) \overset{\Theta}{\rightsquigarrow} (e', \varphi')$.

Lemma 4. *Let $\mathcal{S}$ be a CSP specification, $\mathcal{D}$ a complete derivation of $\mathcal{S}$ performed with the semantics of Fig. 4, and $\mathcal{G}$ the graph produced by $\mathcal{D}$. Then, for each extended rewriting step in $\mathcal{D}$ of the form $(e, \varphi) \overset{\Theta}{\rightsquigarrow} (e', \varphi')$ which is not associated to* (Synchronized Parallelism 4) *we have that a node for $first(e)$ is added to $\mathcal{G}$ with reference φ.*

Proof. We prove the lemma for each rule:

(SKIP), (STOP), (Prefixing), (Process Call), and (Internal Choice 1 and 2) A node for $first(e)$ (in these rules α) is added to $\mathcal{G}$ with reference φ.

(External Choice 1, 2, 3 and 4) and (Synchronized Parallelism 1, 2 and 3) In these rules, the node associated to $first(e)$ (it is α here) could be included or not, depending on whether it has been included by a previous rewriting step. If it is already included, it is due to the specification position of the previous expression in the control is the same as e, and its associated rewriting step or a previous one has added it. If other case, function Ω is called and it includes the node for e with reference φ, since the corresponding φ_P and φ_Q are equal to $(\_, \_)$.

(Hiding 1, 2 and 3) and (Renaming 1, 2 and 3) The proof for these rules is similar to the one for rules of external choice and synchronized parallelism, but here function Σ is called and it adds to $\mathcal{G}$ the node for e with reference φ because the label is distinct of $\bullet$.

(Sequential Composition 1 and 2) In both rules, the node for $first(e)$ (in this case $first(P)$) is included by a rewriting step in Θ. All possible rewriting steps must apply one of these previous rules, and thus, the claim recursively follows.

□

Lemma 5. *Let $\mathcal{S}$ be a CSP specification, $\mathcal{D}$ a complete derivation of $\mathcal{S}$ performed with the semantics of Fig. 4, and $\mathcal{G}$ the graph produced by $\mathcal{D}$. Then, for each rewriting step in $\mathcal{D}$ of the form $e_i \overset{\Theta_i}{\rightsquigarrow} e_{i+1}$ we have that:*

1. *E_c contains an edge $\mathcal{P}os(e_i) \mapsto \mathcal{P}os(first(e'))$ where $e' \overset{\Theta'}{\rightsquigarrow} e'' \in \Theta_i$ and the control can pass from e_i to e', and*
2. *if the control can pass from e_i to e_{i+1} then E_c contains an edge $\mathcal{P}os(e_i) \mapsto \mathcal{P}os(first(e_{i+1}))$.*

Proof. We prove each claim separately:

1. Firstly, we know that Θ cannot be empty. Therefore, rules (SKIP), (STOP), (Process Call), (Prefixing), (Internal Choice 1 and 2) and (Synchronized Parallelism 4) could not be applied. Moreover, (Sequential Composition) could never be applied because if e_i is of the form $P;Q$, then the control can only pass from $P;Q$ to Q (by Definition 2). And Q can only be in the control of the right state; hence, Q cannot appear in Θ. Then the only applicable rules are (External Choice), (Synchronized Parallelism), (Hiding) or (Renaming).

   Let us extend the rewriting step $e' \overset{\Theta'}{\rightsquigarrow} e'' \in \Theta_i$ with references, so that we have $(e', \varphi') \overset{\Theta'}{\rightsquigarrow} (e'', \varphi'') \in \Theta_i$. First, we have to prove that a node with the specification position of e_i is included in the graph and the reference of its successor node is put in each φ' of Θ_i. In rules (External Choice) and (Synchronized Parallelism) it is done using function Ω. The references associated to the selected branches of the operator must be $(\_, \_)$, i.e., the branches has not been developed until now in the derivation. Otherwise, by Definition 2, there is not possible control flow between e_i and e'. In this case, if the corresponding reference is $(\_, \_)$, then Ω adds to $\mathcal{G}$ the specification position of e_i and the reference of the successor node is put in all possible φ'.
   (Hiding) and (Renaming) are analogous but, in these cases, function Σ is used. If the previous rewriting step in the derivation has different position to e_i, then function Σ is called and the node is included in the graph and the reference to the successor is put in φ'. Next rewriting step cannot include again the node for the position of e_i because the label of the operator will be $\bullet$. Moreover, if the node has been already included, then the control cannot pass from e_i to e'. Then, we have to prove that the node associated to $first(e')$ is included in the graph with reference φ'. It is trivially proven by using Lemma 4. Note that Lemma 4 excludes rule (Synchronized Parallelism 4) but in this case both branches must be already in $\mathcal{G}$ by a previous application of (Synchronized Parallelism 1, 2 or 3).
2. In this case, e_i cannot be a `SKIP` neither a `STOP`, because the control cannot pass from them to expression e_{i+1} ($\top$ or $\bot$, respectively) by Definition 2. Expression e_i can neither be a parallelism, a hiding or a renaming because the control cannot pass to e_{i+1} (itself or $\top$).
   If e_i is an external choice we have two possibilities. If we apply (External Choice 1 or 2) then e_i and e_{i+1} have the same specification position and

thus, by Definition 2, no control flow is possible. If we apply (External Choice 3 or 4) the control cannot pass from e_i to e_{i+1}, because e_{i+1} is different to $first(e_i.1)$ or $first(e_i.2)$. This is due to the fact that the nodes associated to these positions have necessarily been added to $\mathcal{G}$ by the rewriting step Θ_i or by a previous rewriting step on derivation $\mathcal{D}$. Therefore, expression e_i must be a process call, a prefixing, an internal choice, or a sequential composition. If it is a sequential composition, rule (Sequential Composition 1) cannot be applied because in this case e_i and e_{i+1} have the same specification position. Therefore, only (Sequential Composition 2) can be applied.
We now prove that the application of any of remaining rules (Process Call), (Prefixing), (Internal Choice 1 and 2), and (Sequential Composition 2) satisfies the property.
For convenience, we extend the rewriting step to also include the references of the graph: $(e_i, \varphi_i) \overset{\Theta_i}{\rightsquigarrow} (e_{i+1}, \varphi_{i+1})$. In all the rules, a node labeled α is added to $\mathcal{G}$ (except in (Prefixing) where is β) and the position of its successor is placed as φ_{i+1}. Furthermore, we know by Lemma 4 that a node for $\mathcal{P}os(first(e_{i+1}))$ is included in the next rewriting step in the derivation $(e_{i+1}, \varphi_{i+1}) \overset{\Theta_{i+1}}{\rightsquigarrow} (e_{i+2}, \varphi_{i+2})$ having associated position φ_{i+1}.
Note that Lemma 4 excludes rule (Synchronized Parallelism 4) but in this case both branches must be already in $\mathcal{G}$ by a previous application of (Synchronized Parallelism 1, 2 or 3).

□

Lemma 6. *Let $\mathcal{S}$ be a CSP specification, $\mathcal{D}$ a derivation of $\mathcal{S}$ performed with the semantics of Fig. 4, and $\mathcal{G}$ the graph produced by $\mathcal{D}$. Then, there exists a synchronization edge $(a \leftrightarrow a')$ in $\mathcal{G}$ for each synchronization in $\mathcal{D}$ where a and a' are the synchronized events.*

Proof. We prove this lemma by induction on the length of the derivation $\mathcal{D} = e_0 \overset{\Theta_0}{\rightsquigarrow} e_1 \overset{\Theta_1}{\rightsquigarrow} \dots \overset{\Theta_n}{\rightsquigarrow} e_{n+1}$. We can assume that the derivation starts with the initial configuration $(\texttt{MAIN}, \emptyset, (0, Root), \emptyset)$, thus in the base case, the only rule applicable is (Process Call) and hence no synchronization is possible. We assume as the induction hypothesis that there exists a synchronization edge $(a \leftrightarrow a') \in \mathcal{G}$ for each synchronization in $e_0 \overset{\Theta_*}{\rightsquigarrow} e_i$ with $0 < i \leq n$ and prove that the lemma also holds for the next rewriting step $e_i \overset{\Theta_i}{\rightsquigarrow} e_{i+1}$.

Firstly, only (Synchronized Parallelism 3) allows the synchronization of events. Therefore, only if e is a synchronizing parallelism, or if a (Synchronized Parallelism 3) is applied in Θ_i, $(a \leftrightarrow a') \in \mathcal{G}$. Then, let us consider the case where $\overset{\Theta_i}{\rightsquigarrow}$ is the application of rule (Synchronized Parallelism 3). This proof is also valid to the case where (Synchronized Parallelism 3) is applied in Θ_i. We have the following rewriting step:

$$\frac{RewritingStep_P \qquad RewritingStep_Q}{(P \underset{X}{\|}_{(\alpha,\varphi_P,\varphi_Q)} Q, G, \varphi, \_) \xrightarrow{a} (P' \underset{X}{\|}_{(\alpha,\varphi'_P,\varphi'_Q)} Q', G', \varphi, \Delta_P \cup \Delta_Q)}$$

$$a \in X \ \wedge \ G' = G_P \cup G_Q \cup \{s_P \overset{a}{\leftrightarrow} s_Q \mid s_P \in \Delta_P \wedge s_Q \in \Delta_Q\} \ \wedge$$

$$(G_{P_\Omega}, \varphi_{P_\Omega}) \ = \ \Omega(G, \varphi_P, \varphi, \alpha) \ \wedge \ (G_{Q_\Omega}, \varphi_{Q_\Omega}) \ = \ \Omega(G, \varphi_Q, \varphi, \alpha)$$

$$\wedge \ \mathit{RewritingStep}_P = (P, G_{P_\Omega}, \varphi_{P_\Omega}, \_) \xrightarrow{a} (P', G_P, \varphi'_P, \Delta_P)$$

$$\wedge \ \mathit{RewritingStep}_Q = (Q, G_{Q_\Omega}, \varphi_{Q_\Omega}, \_) \xrightarrow{a} (Q', G_Q, \varphi'_Q, \Delta_Q)$$

Because (Prefixing) is the only rule that performs an a event without further conditions, we know that P must be a prefixing operator or an expression containing a prefixing operator whose prefix is a, i.e., we know that the rule applied in $\mathit{RewritingStep}_P$ is fired with an event a; and we know that all the rules of the semantics except (Prefixing) need to fire another rule with an event a as a condition. Therefore, at the top of the condition rules, there must be a (Prefixing). The same happens with Q. Hence, two prefixing rules (one for P and one for Q) have been fired as a condition of this rule.

In addition, the new graph G' contains the synchronizations set $\{s_P \overset{a}{\leftrightarrow} s_Q \mid s_P \in \Delta_P \wedge s_Q \in \Delta_Q\}$ where Δ_P and Δ_Q are the sets of references to the events that must synchronize in $\mathit{RewritingStep}_P$ and $\mathit{RewritingStep}_Q$, respectively.

Hence, we have to prove that all and only the events (a) that must synchronize in $\mathit{RewritingStep}_P$ are in Δ_P. We prove this by showing that all references to the synchronized events are propagated down by all rules from the prefixing in the top to the (Synchronized Parallelism 3). And the proof is analogous for $\mathit{RewritingStep}_Q$.

The possible rules applied in $(P, G_{P_\Omega}, \varphi_{P_\Omega}, \_) \xrightarrow{a} (P', G_P, \varphi'_P, \Delta_P)$ are: (Prefixing) In this case, the prefix a is added to Δ_P. (External Choice 3), (External Choice 4), (Sequential Composition 1), (Synchronized Parallelism 1), (Synchronized Parallelism 2), (Hiding 1), (Hiding 2), (Renaming 1), (Renaming 2) In these cases, the set Δ is propagated down. (Synchronized Parallelism 3) In this case, the sets Δ_P and Δ_Q are joined and propagated down.

Therefore, all the synchronized events are in the set Δ_P and the claim follows. □

Theorem 2 (Correctness). *Let $\mathcal{S}$ be a CSP specification, $\mathcal{D}$ a derivation of $\mathcal{S}$ performed with the semantics of Fig. 4, and $\mathcal{G}$ the graph produced by $\mathcal{D}$. Then, $\mathcal{G}$ is the trace associated to $\mathcal{D}$.*

Proof. In order to prove that $\mathcal{G}$ is a trace, we need to prove that it satisfies the properties of Definition 4. For each $e \overset{\Theta}{\rightsquigarrow} e' \in \mathcal{D}$ and for all rewriting steps in Θ we have

- by Lemma 1, if e is a prefixing $(a \rightarrow P)$ and $\mathcal{P}os(P) \in N$, then E_c contains an edge $\mathcal{P}os(a) \mapsto \mathcal{P}os(\rightarrow)$;
- by Lemma 3, if e is a sequential composition $(Q; P)$ and $\mathcal{P}os(P) \in N$, then E_c contains an edge $\mathcal{P}os(\mathit{last}'(Q)) \mapsto \mathcal{P}os(;)$;

- by Lemma 5, if the control can pass from e to e'' where $e'' \overset{\Theta'}{\rightsquigarrow} e''' \in \Theta$, then E_c contains an edge $\mathcal{P}os(e) \mapsto \mathcal{P}os(\mathit{first}(e''))$; and if the control can pass from e to e' then E_c contains an edge $\mathcal{P}os(e) \mapsto \mathcal{P}os(\mathit{first}(e'))$; and
- by Lemma 6, E_s contains a synchronization edge $a \leftrightarrow a'$ for each synchronization occurring in the rewriting step where a and a' are the synchronized events.

Moreover, we know that the only nodes in N are the nodes induced by E_c and E_s because all the nodes inserted in $\mathcal{G}$ are inserted by connecting the new node to the last inserted node (i.e., if the current reference is (q, p), the new node is always inserted as $G[q \overset{p}{\underset{r}{\mapsto}} \alpha]$). Hence, all nodes are related by control or synchronization edges and thus the claim holds. □