A Semantics to Generate the Context-sensitive Synchronized Control-Flow Graph (extended)

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. The CSP language allows the specification and verification of complex concurrent systems. Many analyses for CSP exist that have been successfully applied in different industrial projects. However, the cost of the analyses performed is usually very high, and sometimes prohibitive, due to the complexity imposed by the non-deterministic execution order of processes and to the restrictions imposed on this order by synchronizations. In this work, we define a data structure that allows us to statically simplify a specification before the analyses. This simplification can drastically reduce the time needed by many CSP analyses. We also introduce an algorithm able to automatically generate this data structure from a CSP specification. The algorithm has been proved correct and its implementation for the CSP's animator ProB is publicly available.

1 Introduction

The *Communicating Sequential Processes* (CSP) [3, 12] language allows us to specify complex systems with multiple interacting processes. The study and transformation of such systems often implies different analyses (e.g., deadlock analysis [5], reliability analysis [4], refinement checking [11], etc.) which are often based on a data structure able to represent all computations of a specification.

Recently, a new data structure called *Context-sensitive Synchronized Control-Flow Graph* (CSCFG) has been proposed [7]. This data structure is a graph that allows us to finitely represent possibly infinite computations, and it is particularly interesting because it takes into account the context of process calls, and thus it allows us to produce analyses that are very precise. In particular, some analyses (see, e.g., [8, 9]) use the CSCFG to simplify a specification with respect to some term by discarding those parts of the specification that cannot be executed before the term and thus they cannot influence it. This simplification is automatic and thus it is very useful as a preprocessing stage of other analyses.

However, computing the CSCFG 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 the reason why there does not exist any correctness result which formally relates the CSCFG of a specification to its execution. This result is needed to prove important properties (such as correctness and completeness) of the techniques based on the CSCFG.

In this work, we formally define the CSCFG and a technique to produce the CSCFG of a given CSP specification. Roughly, we instrument the CSP standard semantics (Chapter 7 in [12]) in such a way that the execution of the instrumented semantics produces as a side-effect the portion of the CSCFG associated with the performed computation. Then, we define an algorithm which uses the instrumented semantics to build the complete CSCFG associated with a CSP specification. This algorithm executes the semantics several times to explore all possible computations of the specification, producing incrementally the final CSCFG.

2 The Syntax and Semantics of CSP

In order to make the paper self-contained, this section recalls CSP's syntax and semantics [3, 12]. For concretion, and to facilitate the understanding of the following definitions and algorithm, we have selected a subset of CSP that is sufficiently expressive to illustrate the method, and it contains the most important operators that produce the challenging problems such as deadlocks, non-determinism and parallel execution.

We use the following domains: process names ($M, N \ldots \in Names$), processes ($P, Q \ldots \in Procs$) and events ($a, b \ldots \in \Sigma$). A CSP specification is a finite set of process definitions $N = P$ with $P = M \mid a \rightarrow P \mid P \sqcap Q \mid P \,\Box\, Q \mid P \underset{X \subseteq \Sigma}{||} Q \mid STOP$. Therefore, processes can be a call to another process or a combination of the following operators:

Prefixing ($a \rightarrow P$) Event a must happen before process P.

Internal choice ($P \sqcap Q$) The system chooses non-deterministically to execute one of the two processes P or Q.

External choice ($P \,\Box\, Q$) It is identical to internal choice but the choice comes from outside the system (e.g., the user).

Synchronized parallelism ($P \underset{X \subseteq \Sigma}{||} Q$) Both processes are executed in parallel with a set X of synchronized events. In absence of synchronizations both processes can execute in any order. Whenever a synchronized event $a \in X$ happens in one of the processes, it must also happen in the other at the same time. Whenever the set of synchronized events is not specified, it is assumed that processes are synchronized in all common events. A particular case of parallel execution is *interleaving* (represented by $|||$) where no synchronizations exist (i.e., $X = \emptyset$).

Stop ($STOP$) Synonym of deadlock: It finishes the current process.

We now recall the standard operational semantics of CSP as defined by Roscoe [12]. It is presented in Fig. 1 as a logical inference system. A *state* of the semantics is a process to be evaluated called the *control*. In the following, we assume that the system starts with an initial state `MAIN`, 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 $\Sigma^\tau = \Sigma \cup \{\tau\}$. Events in Σ are visible from the external environment, and can only happen with its co-operation (e.g., actions of the user). Event τ is an internal event that cannot be observed from outside the system and it happens automatically as defined by the semantics. In order to perform computations, we construct an initial state and (non-deterministically) apply the rules of Fig. 1.

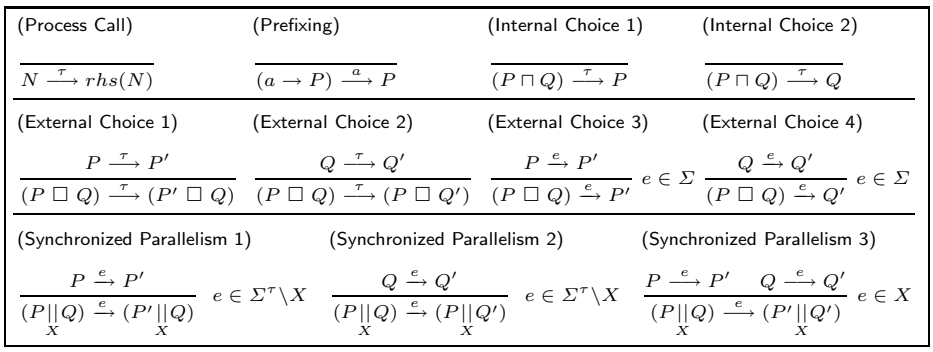


Fig. 1. CSP's operational semantics

3 Context-sensitive Synchronized Control-Flow Graphs

The CSCFG was proposed in [7, 9] as a data structure able to finitely represent all possible (often infinite) computations of a CSP specification. This data structure is particularly useful to simplify a CSP specification before its static analysis. The simplification of industrial CSP specifications allows us to drastically reduce the time needed to perform expensive analyses such as model checking. Algorithms to construct CSCFGs have been implemented [8] and integrated into the most advanced CSP environment ProB [6]. In this section we introduce a new formalization of the CSCFG that directly relates the graph construction to the control-flow of the computations it represents.

A CSCFG is formed by the sequence of expressions that are evaluated during an execution. These expressions are conveniently connected to form a graph. In addition, the source position (in the specification) of each literal (i.e., events, operators and process names) is also included in the CSCFG. This is very useful because it provides the CSCFG with the ability to determine what parts of the source code have been executed and in what order. The inclusion of source positions in the CSCFG implies an additional level of complexity in the semantics, but the benefits of providing the CSCFG with this additional information are clear and, for some applications, essential. Therefore, we use labels (that we call *specification positions*) to identify each literal in a specification which roughly

corresponds to nodes in the CSP specification's abstract syntax tree. We define a function $\mathcal{P}os$ to obtain the specification position of an element of a CSP specification and it is defined over nodes of an abstract syntax tree for a CSP specification. Formally,

Definition 1. *(Specification position) A* specification position *is a pair* (N, w) *where* $N \in \mathcal{N}$ *and* w *is a sequence of natural numbers (we use* Λ *to denote the empty sequence). We let* $\mathcal{P}os(o)$ *denote the specification position of an expression* o*. Each process definition* $N = P$ *of a CSP specification is labelled with specification positions. The specification position of its left-hand side is* $\mathcal{P}os(N) = (N, 0)$*. The right-hand side (abbrev.* rhs*) is labelled with the call* $\texttt{AddSpPos}(P, (N, \Lambda))$*; where function* `AddSpPos` *is defined as follows:*

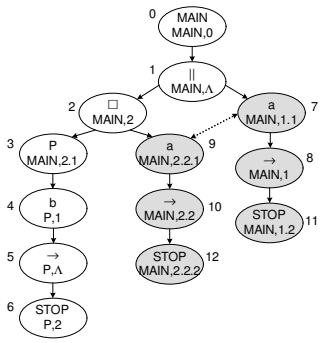
$$\texttt{AddSpPos}(P,(N,w)) = \begin{cases} P_{(N,w)} & \textit{if } \; P \in \mathcal{N} \\ STOP_{(N,w)} & \textit{if } \; P = STOP \\ a_{(N,w.1)} \rightarrow_{(N,w)} \texttt{AddSpPos}(Q,(N,w.2)) & \textit{if } \; P = a \rightarrow Q \\ \texttt{AddSpPos}(Q,(N,w.1)) \; op_{(N,w)} \; \texttt{AddSpPos}(R,(N,w.2)) & \\ \qquad \textit{if } \; P = Q \; op \; R \;\; \forall \; op \in \{\sqcap, \Box, ||\} & \end{cases}$$

We often use $\mathcal{P}os(\mathcal{S})$ *to denote a set with all positions in a specification* $\mathcal{S}$*.*

Example 1. Consider the CSP specification in Fig. 2(a) where literals are labelled with their associated specification positions (they are underlined) so that labels are unique.

$\texttt{MAIN}_{\underline{(\texttt{MAIN},0)}} = (\texttt{a}_{\underline{(\texttt{MAIN},1.1)}} \rightarrow_{\underline{(\texttt{MAIN},1)}} \texttt{STOP}_{\underline{(\texttt{MAIN},1.2)}}) \parallel_{\{\texttt{a}\}}^{\underline{(\texttt{MAIN},\Lambda)}}$
$(\texttt{P}_{\underline{(\texttt{MAIN},2.1)}} \Box_{\underline{(\texttt{MAIN},2)}} (\texttt{a}_{\underline{(\texttt{MAIN},2.2.1)}} \rightarrow_{\underline{(\texttt{MAIN},2.2)}} \texttt{STOP}_{\underline{(\texttt{MAIN},2.2.2)}}))$
$\texttt{P}_{\underline{(\texttt{P},0)}} = \texttt{b}_{\underline{(\texttt{P},1)}} \rightarrow_{\underline{(\texttt{P},\Lambda)}} \texttt{SKIP}_{\underline{(\texttt{P},2)}}$

(a) CSP specification



(b) CSCFG

Fig. 2. CSP specification and its associated CSCFG

In the following, specification positions will be represented with greek letters $(\alpha, \beta, \ldots)$ and we will often use indistinguishably an expression and its associated specification position when it is clear from the context (e.g., in Example 1 we will refer to $(\texttt{P}, 1)$ as $\texttt{b}$).

In order to introduce the definition of CSCFG, we need first to define the concepts of *control-flow*, *path* and *context*.

Definition 2. *(Control-flow) Given a CSP specification $\mathcal{S}$, the* control-flow *is a transitive relation between the specification positions of $\mathcal{S}$. Given two specification positions α, β in $\mathcal{S}$, we say that the* control *of α* can pass *to β iff*

i) $\alpha = N \ \wedge \ \beta = \mathit{first}((N, \Lambda))$ with $N = rhs(N) \in \mathcal{S}$
ii) $\alpha \in \{\sqcap, \square, ||\} \ \wedge \ \beta \in \{\mathit{first}(\alpha.1), \mathit{first}(\alpha.2)\}$
iii) $\alpha = \beta.1 \ \wedge \ \beta = \ \rightarrow$
iv) $\alpha = \ \rightarrow \ \wedge \ \beta = \mathit{first}(\alpha.2)$

where $\mathit{first}(\alpha)$ is defined as follows: $\mathit{first}(\alpha) = \begin{cases} \alpha.1 & \text{if } \alpha = \ \rightarrow \\ \alpha & \text{otherwise} \end{cases}$

We say that a specification position α is executable *in $\mathcal{S}$ iff the control can pass from the initial state (i.e.,* `MAIN`*) to α.*

For instance, in Example 1, the control can pass from $(\texttt{MAIN}, 2.1)$ to $(\texttt{P}, 1)$ due to rule i), from $(\texttt{MAIN}, 2)$ to $(\texttt{MAIN}, 2.1)$ and $(\texttt{MAIN}, 2.2.1)$ due to rule ii), from $(\texttt{MAIN}, 2.2.1)$ to $(\texttt{MAIN}, 2.2)$ due to rule iii), and from $(\texttt{MAIN}, 2.2)$ to $(\texttt{MAIN}, 2.2.2)$ due to rule iv).

As we will work with graphs whose nodes are labelled with positions, we use $l(n)$ to refer to the label of node n.

Definition 3. *(Path) Given a labelled graph $\mathcal{G} = (N, E)$, a* path *between two nodes n_1, $m \in N$, $Path(n_1, m)$, is a sequence $n_1, \ldots, n_k$ such that $n_k \mapsto m \in E$ and for all $1 \leq i < k$ we have $n_i \mapsto n_{i+1} \in E$. The path is* loop-free *if for all $i \neq j$ we have $n_i \neq n_j$.*

Definition 4. *(Context) Given a labelled graph $\mathcal{G} = (N, E)$ and a node $n \in N$, the* context *of n, $\mathcal{C}on(n) = \{m \mid l(m) = M$ with $(M = P) \in \mathcal{S}$ and there exists a loop-free path $m \mapsto^* n\}$.*

Intuitively speaking, the context of a node represents the set of processes in which a particular node is being executed. This is represented by the set of process calls in the computation that were done before the specified node. For instance, the CSCFG associated with the specification in Example 1 is shown in Fig. 2(b). In this graph we have that $\mathcal{C}on(4) = \{0, 3\}$, i.e., `b` is being executed after having called processes `MAIN` and `P`. Note that focussing on a process call node we can use the context to identify loops; i.e., we have a loop whenever $n \in \mathcal{C}on(m)$ with $l(n) = l(m) \in Names$. Note also that the CSCFG is unique for a given CSP specification [9].

Definition 5. *(Context-sensitive Synchronized Control-Flow Graph) Given a CSP specification $\mathcal{S}$, its* Context-sensitive Synchronized Control-Flow Graph *(CSCFG) is a labelled directed graph $\mathcal{G} = (N, E_c, E_l, E_s)$ where N is a set of nodes such that $\forall \ n \in N.\ l(n) \in \mathcal{P}os(\mathcal{S})$ and $l(n)$ is executable in $\mathcal{S}$; and edges are divided into three groups:* control-flow edges *(E_c),* loop edges *(E_l) and* synchronization edges *(E_s).*

- *E_c is a set of one-way edges (denoted with $\mapsto$) representing the possible control-flow between two nodes. Control edges do not form loops. The root of the tree formed by E_c is the position of the initial call to* `MAIN`.
- *E_l is a set of one-way edges (denoted with $\rightsquigarrow$) such that $(n_1 \rightsquigarrow n_2) \in E_l$ iff $l(n_1)$ and $l(n_2)$ are (possibly different) process calls that refer to the same process $M \in \mathcal{N}$ and $n_2 \in \mathcal{C}on(n_1)$.*
- *E_s is a set of two-way edges (denoted with $\leftrightarrow$) representing the possible synchronization of two event nodes ($l(n) \in \Sigma$).*
- *Given a CSCFG, every node labelled (M, Λ) has one and only one incoming edge in E_c; and every process call node has one and only one outgoing edge which belongs to either E_c or E_l.*

Example 2. Consider again the specification of Example 1, shown in Fig. 2(a), and its associated CSCFG, shown in Fig. 2(b). For the time being, the reader can ignore the numbering and color of the nodes; they will be explained in Section 4. Each process call is connected to a subgraph which contains the right-hand side of the called process. For convenience, in this example there are no loop edges; there are control-flow edges and one synchronization edge between nodes $(\texttt{MAIN}, 2.2.1)$ and $(\texttt{MAIN}, 1.1)$ representing the synchronization of event `a`.

Note that the CSCFG shows the exact processes that have been evaluated with an explicit causality relation; and, in addition, it shows the specification positions that have been evaluated and in what order. Therefore, it is not only useful as a program comprehension tool, but it can be used for program simplification. For instance, with a simple backwards traversal from `a`, the CSCFG reveals that the only part of the code that can be executed before `a` is the underlined part:

$$\underline{\texttt{MAIN}} = (\underline{\texttt{a}} \rightarrow \texttt{STOP}) \underset{\underline{\{\texttt{a}\}}}{\parallel} (\texttt{P} \,\underline{\Box}\, (\underline{\texttt{a}} \rightarrow \texttt{STOP}))$$
$$\texttt{P} = \texttt{b} \rightarrow \texttt{STOP}$$

Hence, the specification can be significantly simplified for those analyses focussing on the occurrence of event `a`.

4 An Algorithm to Generate the CSCFG

This section introduces an algorithm able to generate the CSCFG associated with a CSP specification. The algorithm uses an instrumented operational semantics of CSP which (i) generates as a side-effect the CSCFG associated with the computation performed with the semantics; (ii) it controls that no infinite loops are executed; and (iii) it ensures that the execution is deterministic.

Algorithm 1 controls that the semantics is executed repeatedly in order to deterministically execute all possible computations—of the original (non-deterministic) specification—and the CSCFG for the whole specification is constructed incrementally with each execution of the semantics. The key point of

the algorithm is the use of a stack that records the actions that can be performed by the semantics. In particular, the stack contains tuples of the form $(rule, rules)$ where $rule$ indicates the rule that must be selected by the semantics in the next execution step, and $rules$ is a set with the other possible rules that can be selected. The algorithm uses the stack to prepare each execution of the semantics indicating the rules that must be applied at each step. For this, function `UpdStack` is used; it basically avoids to repeat the same computation with the semantics. When the semantics finishes, the algorithm prepares a new execution of the semantics with an updated stack. This is repeated until all possible computations are explored (i.e., until the stack is empty).

The standard operational semantics of CSP [12] can be non-terminating due to infinite computations. Therefore, the instrumentation of the semantics incorporates a loop-checking mechanism to ensure termination.

Algorithm 1 General Algorithm

Build the initial state of the semantics: $state = (\texttt{MAIN}_{(\texttt{MAIN},0)}, \emptyset, \bullet, (\emptyset, \emptyset), \emptyset, \emptyset)$
repeat
  repeat
    Run the rules of the instrumented semantics with the state $state$
  until no more rules can be applied
  Get the new state: $state = (\_, G, \_, (\emptyset, S_0), \_, \zeta)$
  $state = (\texttt{MAIN}_{(\texttt{MAIN},0)}, G, \bullet, (\texttt{UpdStack}(S_0), \emptyset), \emptyset, \emptyset)$
until $\texttt{UpdStack}(S_0) = \emptyset$
return G
where function `UpdStack` is defined as follows:

$$\texttt{UpdStack}(S) = \begin{cases} (rule, rules \backslash \{rule\}) : S' & \text{if } S = (\_, rules) : S' \text{ and } rule \in rules \\ \texttt{UpdStack}(S') & \text{if } S = (\_, \emptyset) : S' \\ \emptyset & \text{if } S = \emptyset \end{cases}$$

The instrumented semantics used by Algorithm 1 is shown in Fig. 3. It is an operational semantics where we assume that every literal in the specification has been labelled with its specification position (denoted by a subscript, e.g., P_α). In this semantics, a *state* is a tuple $(P, G, m, (S, S_0), \Delta, \zeta)$, where P is the process to be evaluated (the *control*), G is a directed graph (i.e., the CSCFG constructed so far), m is a numeric reference to the current node in G, (S, S_0) is a tuple with two stacks (where the empty stack is denoted by $\emptyset$) that contains the rules to apply and the rules applied so far, Δ is a set of references to nodes used to draw synchronizations in G and ζ is a graph like G, but it only contains the part of the graph generated for the current computation, and it is used to detect loops. The basic idea of the graph construction is to record the current control with a fresh reference[1] n by connecting it to its parent m. We use the notation $G[n \overset{m}{\mapsto} \alpha]$ either to introduce a node in G or as a condition on G (i.e., G contains node n). This node has reference n, is labelled with specification position α and its

[1] We assume that fresh references are numeric and generated incrementally.

parent is m. The edge introduced can be a control, a synchronization or a loop edge. This notation is very convenient because it allows us to add nodes to G, but also to extract information from G. For instance, with $G[3 \overset{m}{\mapsto} \alpha]$ we can know the parent of node 3 (the value of m), and the specification position of node 3 (the value of α).

Note that the initial state for the semantics used by Algorithm 1 has $\texttt{MAIN}_{(\texttt{MAIN},0)}$ in the control. This initial call to `MAIN` does not appear in the specification, thus we label it with a special specification position $(\texttt{MAIN}, 0)$ which is the root of the CSCFG (see Fig. 2(b)). Note that we use $\bullet$ as a reference in the initial state. The first node added to the CSCFG (i.e., the root) will have parent reference $\bullet$. Therefore, here $\bullet$ denotes the empty reference because the root of the CSCFG has no parent.

An explanation for each rule of the semantics follows.

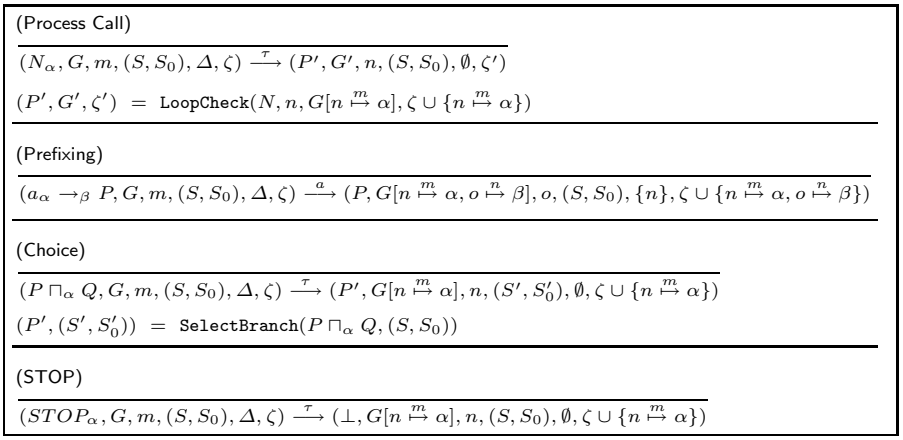


Fig. 3. An instrumented operational semantics that generates the CSCFG

(Process Call) The called process N is unfolded, node n (a fresh reference) is added to the graphs G and ζ with specification position α and parent m. In the new state, n represents the current reference. The new expression in the control is P', computed with function `LoopCheck` which is used to prevent infinite unfolding and is defined below. No event can synchronize in this rule, thus Δ is empty.

$$\texttt{LoopCheck}(N, n, G, \zeta) = \begin{cases} (\circlearrowleft_s(rhs(N)), G[n \rightsquigarrow s], \zeta \cup \{n \rightsquigarrow s\}) & \text{if } \exists s.s \overset{t}{\mapsto} N \in G \\ & \wedge s \in Path(0, n) \\ (rhs(N), G, \zeta) & \text{otherwise} \end{cases}$$

Function `LoopCheck` checks whether the process call in the control has not been already executed (if so, we are in a loop). When a loop is detected, the right-

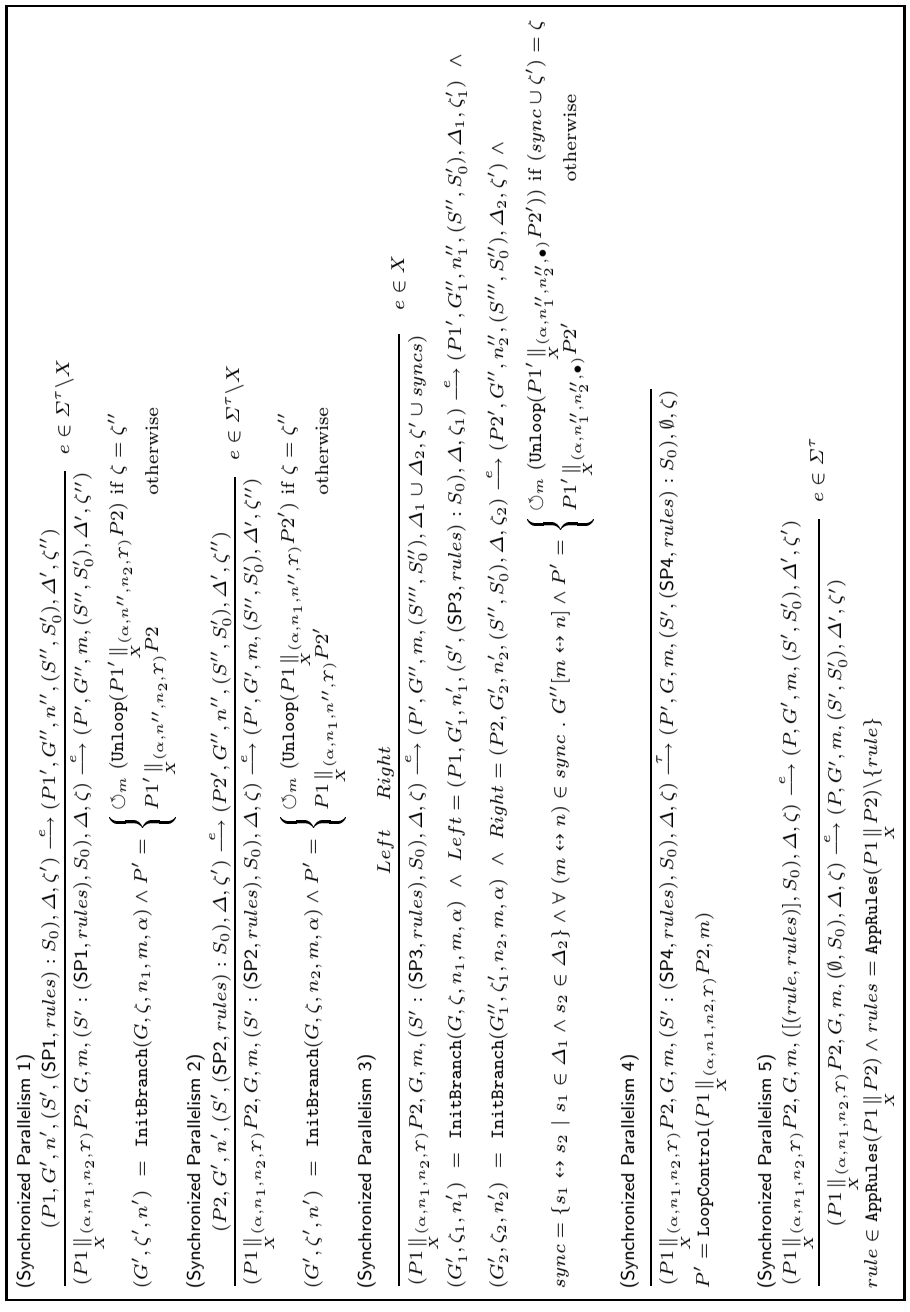


Fig. 3. An instrumented operational semantics that generates the CSCFG (cont.)

hand side of the called process is labelled with a special symbol $\circlearrowleft_s$ and a loop edge between nodes n and s is added to the graph. The loop symbol $\circlearrowleft$ is labelled with the position s of the process call of the loop. This label is later used by rule (Synchronized Parallelism 4) to decide whether the process must be stopped. It is also used to know what is the reference of the process' node if it is unfolded again.

(Prefixing) This rule adds nodes n (the prefix) and o (the prefixing operator) to the graphs G and ζ. In the new state, o becomes the current reference. The new control is P. The set Δ is $\{n\}$ to indicate that event a has occurred and it must be synchronized when required by (Synchronized Parallelism 3).

(Choice) The only sources of non-determinism are choice operators (different branches can be selected for execution) and parallel operators (different order of branches can be selected for execution). Therefore, every time the semantics executes a choice or a parallelism, they are made deterministic thanks to the information in the stack S. Both internal and external can be treated with a single rule because the CSCFG associated to a specification with external choices is identical to the CSCFG associated to the specification with the external choices replaced by internal choices. This rule adds node n to the graphs which is labelled with the specification position α and has parent m. In the new state, n becomes the current reference. No event can synchronize in this rule, thus Δ is empty.

Function `SelectBranch` is used to produce the new control P' and the new tuple of stacks (S', S'_0), by selecting a branch with the information of the stack. Note that, for simplicity, the lists constructor ":" has been overloaded, and it is also used to build lists of the form $(A : a)$ where A is a list and a is the last element:

$$\texttt{SelectBranch}(P \sqcap_\alpha Q, (S, S_0)) = \begin{cases} (P, (S', (\mathsf{C1}, \{\mathsf{C2}\}) : S_0)) & \text{if } S = S' : (\mathsf{C1}, \{\mathsf{C2}\}) \\ (Q, (S', (\mathsf{C2}, \emptyset) : S_0)) & \text{if } S = S' : (\mathsf{C2}, \emptyset) \\ (P, (\emptyset, (\mathsf{C1}, \{\mathsf{C2}\}) : S_0)) & \text{otherwise} \end{cases}$$

If the last element of the stack S indicates that the first branch of the choice ($\mathsf{C1}$) must be selected, then P is the new control. If the second branch must be selected ($\mathsf{C2}$), the new control is Q. In any other case the stack is empty, and thus this is the first time that this choice is evaluated. Then, we select the first branch (P is the new control) and we add $(\mathsf{C1}, \{\mathsf{C2}\})$ to the stack S_0 indicating that $\mathsf{C1}$ has been fired, and the remaining option is $\mathsf{C2}$.

For instance, when the CSCFG of Fig. 2(b) is being constructed and we reach the choice operator (i.e., $(\texttt{MAIN}, 2)$), then the left branch of the choice is evaluated and $(\mathsf{C1}, \{\mathsf{C2}\})$ is added to the stack to indicate that the left branch has been evaluated. The second time it is evaluated, the stack is updated to $(\mathsf{C2}, \emptyset)$ and the right branch is evaluated. Therefore, the selection of branches is predetermined by the stack, thus, Algorithm 1 can decide what branches are evaluated by conveniently handling the information of the stack.

(Synchronized Parallelism 1 and 2) The stack determines what rule to use when a parallelism operator is in the control. If the last element in the stack is $\mathsf{SP1}$, then (Synchronized Parallelism 1) is used. If it is $\mathsf{SP2}$, (Synchronized Parallelism 2) is used.

In a synchronized parallelism composition, both parallel processes can be intertwiningly executed until a synchronized event is found. Therefore, nodes for both processes can be added interwoven to the graph. Hence, the semantics needs to know in every state the references to be used in both branches. This is done by labelling each parallelism operator with a tuple of the form $(\alpha, n_1, n_2, \Upsilon)$ where α is the specification position of the parallelism operator; n_1 and n_2 are respectively the references of the last node introduced in the left and right branches of the parallelism, and they are initialised to $\bullet$; and Υ is a node reference used to decide when to unfold a process call (in order to avoid infinite loops), also initialised to $\bullet$. The sets Δ' and ζ' are passed down unchanged so that another rule can use them if necessary.

These rules develop the branches of the parallelism until they are finished or until they must synchronize. They use function `InitBranch` to introduce the parallelism into the graph the first time it is executed and only if it has not been introduced in a previous computation. For instance, consider a state where a parallelism operator is labelled with $((\texttt{MAIN}, \Lambda), \bullet, \bullet, \bullet)$. Therefore, it is evaluated for the first time, and thus, when, e.g., rule (Synchronized Parallelism 1) is applied, a node $1 \overset{0}{\mapsto} (\texttt{MAIN}, \Lambda)$, which refers to the parallelism operator, is added to G and the parallelism operator is relabelled to $((\texttt{MAIN}, \Lambda), \texttt{x}, \bullet, \bullet)$ where x is the new reference associated with the left branch. After executing function `InitBranch`, we get a new graph and a new reference. Its definition is the following:

$$\texttt{InitBranch}(G, \zeta, n, m, \alpha) = \begin{cases} (G[o \overset{m}{\mapsto} \alpha], \zeta \cup \{o \overset{m}{\mapsto} \alpha\}, o) & \text{if } n = \bullet \\ (G, \zeta, n) & \text{otherwise} \end{cases}$$

(Synchronized Parallelism 3) It is applied when the last element in the stack is SP3. It is used to synchronize the parallel processes. In this rule, Υ is replaced by $\bullet$, meaning that a synchronization edge has been drawn and the loops could be unfolded again if it is needed. The set *sync* of all the events that have been executed in this step must be synchronized. Therefore, all the events occurred in the subderivations of $P1$ (Δ_1) and $P2$ (Δ_2) are mutually synchronized and added to both G'' and ζ'. In the case that all the synchronizations occurred in this step are already in ζ', this rule detects that the parallelism is in a loop; and thus, in the new control the parallelism operator is labelled with $\circlearrowleft$ and all the other loop labels are removed from it. This is done by a trivial function `Unloop`.

(Synchronized Parallelism 4) This rule is applied when the last element in the stack is SP4. It is used when none of the parallel processes can proceed (because they already finished, deadlocked or were labelled with $\circlearrowleft$). When a process is labelled as a loop with $\circlearrowleft$, it can be unlabelled to unfold it once[2]in order to allow the other processes to continue. This happens when the looped process is in parallel with other process and the later is waiting to synchronize with the former. In order to perform the synchronization, both processes must continue, thus the loop is unlabelled. Hence, the system must stop only when both parallel processes are marked as a loop. This task is done by function `LoopControl`.

[2] Only once because it will be labelled again by rule (Process Call) when the loop is repeated.

It decides if the branches of the parallelism should be further unfolded or they should be stopped (e.g., due to a deadlock or an infinite loop):

$$\texttt{LoopControl}(P \underset{X}{\|}_{(\alpha,p,q,\Upsilon)} Q, m) =$$

$$\begin{cases} \circlearrowleft_m(P'_{\circlearrowleft} \underset{X}{\|}_{(\alpha,p_{\circlearrowleft},q_{\circlearrowleft},\bullet)} Q'_{\circlearrowleft}) & \text{if } P' = \circlearrowleft_{p_{\circlearrowleft}}(P'_{\circlearrowleft}) \wedge Q' = \circlearrowleft_{q_{\circlearrowleft}}(Q'_{\circlearrowleft}) \\ \circlearrowleft_m(P'_{\circlearrowleft} \underset{X}{\|}_{(\alpha,p_{\circlearrowleft},q',\bullet)} \bot) & \text{if } P' = \circlearrowleft_{p_{\circlearrowleft}}(P'_{\circlearrowleft}) \ \wedge (Q' = \bot \vee (\Upsilon = p_{\circlearrowleft} \wedge Q' \neq \circlearrowleft\_(\_))) \\ P'_{\circlearrowleft} \underset{X}{\|}_{(\alpha,p_{\circlearrowleft},q',p_{\circlearrowleft})} Q' & \text{if } P' = \circlearrowleft_{p_{\circlearrowleft}}(P'_{\circlearrowleft}) \ \wedge Q' \neq \bot \wedge \ Q' \neq \circlearrowleft\_(\_) \wedge \Upsilon \neq p_{\circlearrowleft} \\ \bot & \text{otherwise} \end{cases}$$

where $(P', p', Q', q') \in \{(P, p, Q, q), (Q, q, P, p)\}$.

When one of the branches has been labelled as a loop, there are three options: (i) The other branch is also a loop. In this case, the whole parallelism is marked as a loop labelled with its parent, and Υ is put to $\bullet$. (ii) Either it is a loop that has been unfolded without drawing any synchronization (this is known because Υ is equal to the parent of the loop), or the other branch already terminated (i.e., it is $\bot$). In this case, the parallelism is also marked as a loop, and the other branch is put to $\bot$ (this means that this process has been deadlocked). Also here, Υ is put to $\bullet$. (iii) If we are not in a loop, then we allow the parallelism to proceed by unlabelling the looped branch. In the rest of the cases $\bot$ is returned representing that this is a deadlock, and thus, stopping further computations.

(Synchronized Parallelism 5) This rule is used when the stack is empty. It basically analyses the control and decides what are the applicable rules of the semantics. This is done with function `AppRules` which returns the set of rules R that can be applied to a synchronized parallelism $P \underset{X}{\|} Q$:

$$\texttt{AppRules}(P \underset{X}{\|} Q) = \begin{cases} \{\mathsf{SP1}\} & \text{if } \tau \in \texttt{FstEvs}(P) \\ \{\mathsf{SP2}\} & \text{if } \tau \notin \texttt{FstEvs}(P) \wedge \tau \in \texttt{FstEvs}(Q) \\ R & \text{if } \tau \notin \texttt{FstEvs}(P) \wedge \tau \notin \texttt{FstEvs}(Q) \ \wedge \ R \neq \emptyset \\ \{\mathsf{SP4}\} & \text{otherwise} \end{cases}$$

where

$$\begin{cases} \mathsf{SP1} \in R & \text{if } \exists e \in \texttt{FstEvs}(P) \wedge e \notin X \\ \mathsf{SP2} \in R & \text{if } \exists e \in \texttt{FstEvs}(Q) \wedge e \notin X \\ \mathsf{SP3} \in R & \text{if } \exists e \in \texttt{FstEvs}(P) \ \wedge \exists e \in \texttt{FstEvs}(Q) \wedge e \in X \end{cases}$$

Essentially, `AppRules` decides what rules are applicable depending on the events that could happen in the next step. These events can be inferred by using function `FstEvs`. In particular, given a process P, function `FstEvs` returns the set of events that can fire a rule in the semantics using P as the control. Therefore, rule (Synchronized Parallelism 5) prepares the stack allowing the semantics to proceed with the correct rule.

$\texttt{FstEvs}(P) =$

$$\begin{cases} \{a\} & \text{if } P = a \to Q \\ \emptyset & \text{if } P = \circlearrowleft Q \ \vee \ P = \bot \\ \{\tau\} & \text{if } P = M \vee P = \texttt{STOP} \vee P = Q \sqcap R \vee P = (\bot \underset{X}{\parallel} \bot) \\ & \vee \ P = (\circlearrowleft Q \underset{X}{\parallel} \circlearrowleft R) \ \vee \ P = (\circlearrowleft Q \underset{X}{\parallel} \bot) \ \vee \ P = (\bot \underset{X}{\parallel} \circlearrowleft R) \\ & \vee (P = (\circlearrowleft Q \underset{X}{\parallel} R) \wedge \texttt{FstEvs}(R) \subseteq X) \vee (P = (Q \underset{X}{\parallel} \circlearrowleft R) \wedge \texttt{FstEvs}(Q) \subseteq X) \\ & \vee (P = Q \underset{X}{\parallel} R \wedge \texttt{FstEvs}(Q) \subseteq X \wedge \texttt{FstEvs}(R) \subseteq X \wedge \bigcap_{M \in \{Q,R\}} \texttt{FstEvs}(M) = \emptyset) \\ E & \text{otherwise, where } P = Q \underset{X}{\parallel} R \wedge E = (\texttt{FstEvs}(Q) \cup \texttt{FstEvs}(R)) \setminus \\ & \qquad (X \cap (\texttt{FstEvs}(Q) \setminus \texttt{FstEvs}(R) \cup \texttt{FstEvs}(R) \setminus \texttt{FstEvs}(Q))) \end{cases}$$

(STOP) Whenever this rule is applied, the subcomputation finishes because $\bot$ is put in the control, and this special constructor has no associated rule. A node with the STOP position is added to the graph.

We illustrate this semantics with a simple example.

Example 3. Consider again the specification in Example 1. Due to the choice operator, in this specification two different events can occur, namely `b` and `a`. Therefore, Algorithm 1 obtains two computations, called respectively **First iteration** and **Second iteration** in Fig. 4. In this figure, for each state, we show a sequence of rules applied from left to right to obtain the next state. Here, for clarity, specification positions have been omitted from the control. We first execute the semantics with the initial state $(\texttt{MAIN}_{(\texttt{MAIN},0)}, \emptyset, \bullet, (\emptyset, \emptyset), \emptyset, \emptyset)$ and get the computation **First iteration**. This computation corresponds to the execution of the left branch of the choice (i.e., `P`) with the occurrence of event `b`. The final state is $State\ 6 = (\bot, G_5, 0, (\emptyset, S_6), \emptyset, \emptyset)$. Note that the stack S_6 contains a pair $(\mathsf{C1}, \{\mathsf{C2}\})$ to denote that the left branch of the choice has been executed. Then, the algorithm calls function `UpdStack` and executes the semantics again with the new initial state $State\ 7 = (\texttt{MAIN}_{(\texttt{MAIN},0)}, G_5, \bullet, ([(\mathsf{C2}, \emptyset), (\mathsf{SP2}, \emptyset)], \emptyset), \emptyset, \emptyset)$ and it gets the computation **Second iteration**. After this execution the final CSCFG (G_9) has been computed. Figure 2(b) shows the CSCFG generated where white nodes were generated in the first iteration; and grey nodes were generated in the second iteration.

For those readers interested in the complete sequence of rewriting steps performed by the semantics, we provide in Fig. 5 the complete derivations of the semantics that the algorithm fired. Here, for clarity, specification positions have been omitted from the control and each computation step is labelled with the applied rule.

Next, we show a more interesting example where non-terminating processes appear.

First iteration

$State\ 0 = (\mathtt{MAIN}_{(\mathtt{MAIN},0)}, G_0, \bullet, (\emptyset,\emptyset), \emptyset, \emptyset)$ where $G_0 = \emptyset$ (PC)

$State\ 1 = ((\mathtt{a} \rightarrow \mathtt{STOP}) \underset{\{\mathtt{a}\}}{\parallel}_{((\mathtt{MAIN},\Lambda),\bullet,\bullet,\bullet)} (\mathtt{P}\Box(\mathtt{a} \rightarrow \mathtt{STOP})), G_1, 0, (\emptyset,\emptyset), \emptyset, \emptyset)$
where $G_1 = G_0[0 \mapsto (\mathtt{MAIN}, 0)]$ (SP5)(SP2)(Choice)

$State\ 2 = ((\mathtt{a} \rightarrow \mathtt{STOP}) \underset{\{\mathtt{a}\}}{\parallel}_{((\mathtt{MAIN},\Lambda),\bullet,2,\bullet)} \mathtt{P}, G_2, 0, (\emptyset, S_2), \emptyset, \emptyset)$
where $G_2 = G_1[1 \overset{0}{\mapsto} (\mathtt{MAIN}, \Lambda), 2 \overset{1}{\mapsto} (\mathtt{MAIN}, 2)]$ and $S_2 = [(\mathsf{C1}, \{\mathsf{C2}\}), (\mathsf{SP2}, \emptyset)]$ (SP5)(SP2)(PC)

$State\ 3 = ((\mathtt{a} \rightarrow \mathtt{STOP}) \underset{\{\mathtt{a}\}}{\parallel}_{((\mathtt{MAIN},\Lambda),\bullet,3,\bullet)} (\mathtt{b} \rightarrow \mathtt{STOP}), G_3, 0, (\emptyset, S_3), \emptyset, \emptyset)$
where $G_3 = G_2[3 \overset{2}{\mapsto} (\mathtt{MAIN}, 2.1)]$ and $S_3 = (\mathsf{SP2}, \emptyset) : S_2$ (SP5)(SP2)(Pref)

$State\ 4 = ((\mathtt{a} \rightarrow \mathtt{STOP}) \underset{\{\mathtt{a}\}}{\parallel}_{((\mathtt{MAIN},\Lambda),\bullet,5,\bullet)} \mathtt{STOP}, G_4, 0, (\emptyset, S_4), \{4\}, \emptyset)$
where $G_4 = G_3[4 \overset{3}{\mapsto} (\mathtt{P}, 1), 5 \overset{4}{\mapsto} (\mathtt{P}, \Lambda)]$ and $S_4 = (\mathsf{SP2}, \emptyset) : S_3$ (SP5)(SP2)(STOP)

$State\ 5 = ((\mathtt{a} \rightarrow \mathtt{STOP}) \underset{\{\mathtt{a}\}}{\parallel}_{((\mathtt{MAIN},\Lambda),\bullet,6,\bullet)} \bot, G_5, 0, (\emptyset, S_5), \emptyset, \emptyset)$
where $G_5 = G_4[6 \overset{5}{\mapsto} (\mathtt{P}, 2)]$ and $S_5 = (\mathsf{SP2}, \emptyset) : S_4$ (SP5)(SP4)

$State\ 6 = (\bot, G_5, 0, (\emptyset, S_6), \emptyset, \emptyset)$ where $S_6 = (\mathsf{SP4}, \emptyset) : S_5$
$= [(\mathsf{SP4}, \emptyset), (\mathsf{SP2}, \emptyset), (\mathsf{SP2}, \emptyset), (\mathsf{SP2}, \emptyset), (\mathsf{C1}, \{\mathsf{C2}\}), (\mathsf{SP2}, \emptyset)]$

Second iteration

$State\ 7 = (\mathtt{MAIN}_{(\mathtt{MAIN},0)}, G_5, \bullet, (\mathtt{UpdStack}(S_6), \emptyset), \emptyset, \emptyset) =$
$(\mathtt{MAIN}_{(\mathtt{MAIN},0)}, G_5, \bullet, ([(\mathsf{C2}, \emptyset), (\mathsf{SP2}, \emptyset)], \emptyset), \emptyset, \emptyset)$ (PC)

$State\ 8 = ((\mathtt{a} \rightarrow \mathtt{STOP}) \underset{\{\mathtt{a}\}}{\parallel}_{((\mathtt{MAIN},\Lambda),\bullet,\bullet,\bullet)} (\mathtt{P}\Box(\mathtt{a} \rightarrow \mathtt{STOP})), G_5, 0, (S_8, \emptyset), \emptyset, \emptyset)$
where $S_8 = [(\mathsf{C2}, \emptyset), (\mathsf{SP2}, \emptyset)]$ (SP2)(Choice)

$State\ 9 = ((\mathtt{a} \rightarrow \mathtt{STOP}) \underset{\{\mathtt{a}\}}{\parallel}_{((\mathtt{MAIN},\Lambda),\bullet,2,\bullet)} (\mathtt{a} \rightarrow \mathtt{STOP}), G_5, 0, (\emptyset, S_9), \emptyset, \emptyset)$
where $S_9 = [(\mathsf{C2}, \emptyset), (\mathsf{SP2}, \emptyset)]$ (SP5)(SP3)(Pref)(Pref)

$State\ 10 = (\mathtt{STOP} \underset{\{\mathtt{a}\}}{\parallel}_{((\mathtt{MAIN},\Lambda),8,10,\bullet)} \mathtt{STOP}, G_6, 0, (\emptyset, S_{10}), \{7, 9\}, \{7 \leftrightarrow 9\})$
where $G_6 = G_5[7 \overset{1}{\mapsto} (\mathtt{MAIN}, 1.1), 8 \overset{7}{\mapsto} (\mathtt{MAIN}, 1), 9 \overset{2}{\mapsto} (\mathtt{MAIN}, 2.2.1), 10 \overset{9}{\mapsto} (\mathtt{MAIN}, 2.2)]$
and $S_{10} = (\mathsf{SP3}, \emptyset) : S_9$ (SP5)(SP1)(STOP)

$State\ 11 = (\bot \underset{\{\mathtt{a}\}}{\parallel}_{((\mathtt{MAIN},\Lambda),11,10,\bullet)} \mathtt{STOP}, G_7, 0, (\emptyset, S_{11}), \emptyset, \{7 \leftrightarrow 9\})$
where $G_7 = G_6[11 \overset{8}{\mapsto} (\mathtt{MAIN}, 1.2)]$ and $S_{11} = (\mathsf{SP1}, \emptyset) : S_{10}$ (SP5)(SP2)(STOP)

$State\ 12 = (\bot \underset{\{\mathtt{a}\}}{\parallel}_{((\mathtt{MAIN},\Lambda),11,12,\bullet)} \bot, G_8, 0, (\emptyset, S_{12}), \emptyset, \{7 \leftrightarrow 9\})$
where $G_8 = G_7[12 \overset{10}{\mapsto} (\mathtt{MAIN}, 2.2.2)]$ and $S_{12} = (\mathsf{SP2}, \emptyset) : S_{11}$ (SP5)(SP4)

$State\ 13 = (\bot, G_8, 0, (\emptyset, S_{13}), \emptyset, \{7 \leftrightarrow 9\})$
where $S_{13} = (\mathsf{SP4}, \emptyset) : S_{12} = [(\mathsf{SP4}, \emptyset), (\mathsf{SP2}, \emptyset), (\mathsf{SP1}, \emptyset), (\mathsf{SP3}, \emptyset), (\mathsf{C2}, \emptyset), (\mathsf{SP2}, \emptyset)]$

$State\ 14 = (\mathtt{MAIN}_{(\mathtt{MAIN},0)}, G_8[7 \leftrightarrow 9], \bullet, (\mathtt{UpdStack}(S_{13}), \emptyset), \emptyset, \emptyset) =$
$(\mathtt{MAIN}_{(\mathtt{MAIN},0)}, G_9, \bullet, (\emptyset,\emptyset), \emptyset, \emptyset)$

Fig. 4. An example of computation with Algorithm 1

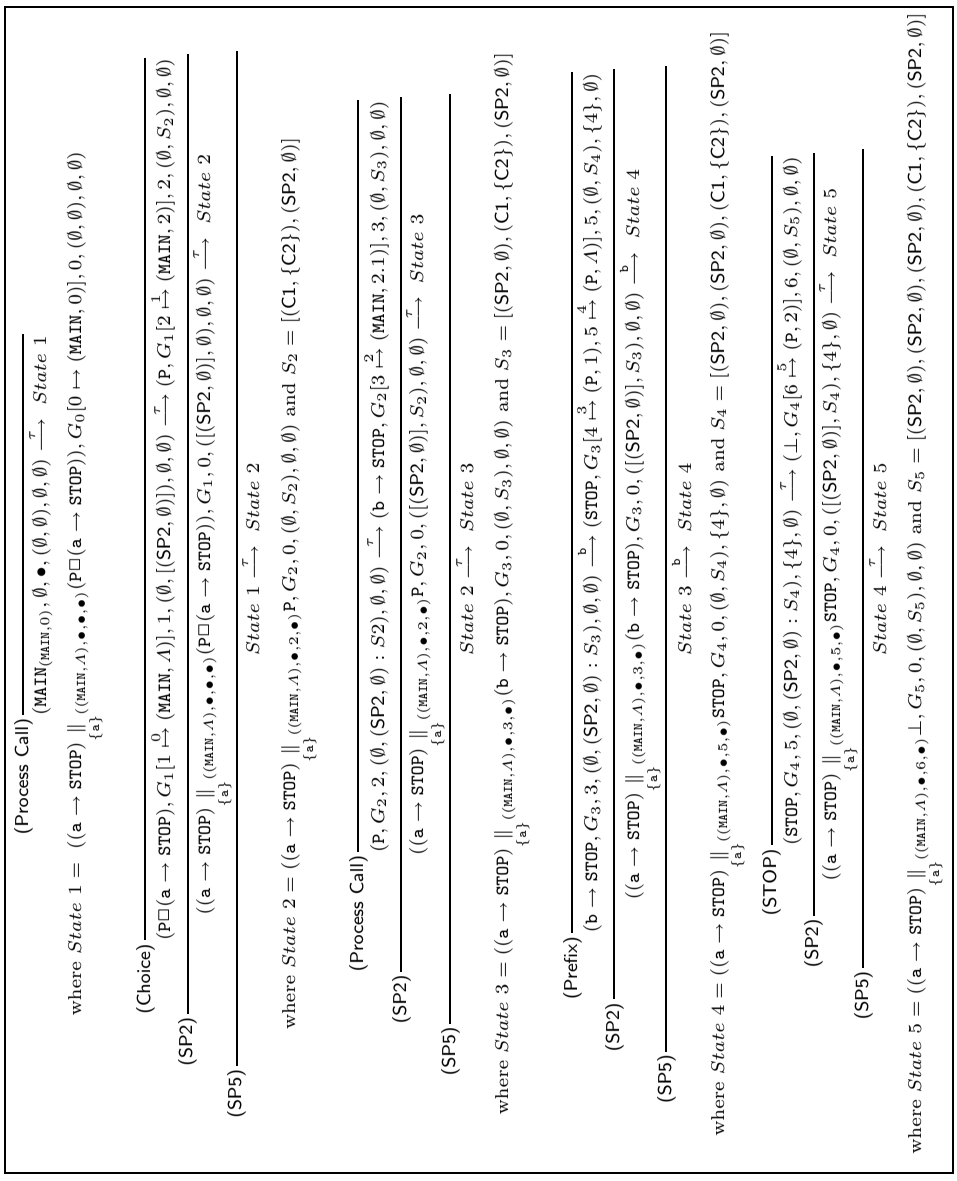


Fig. 5. An example of computation (step by step) with Algorithm 1

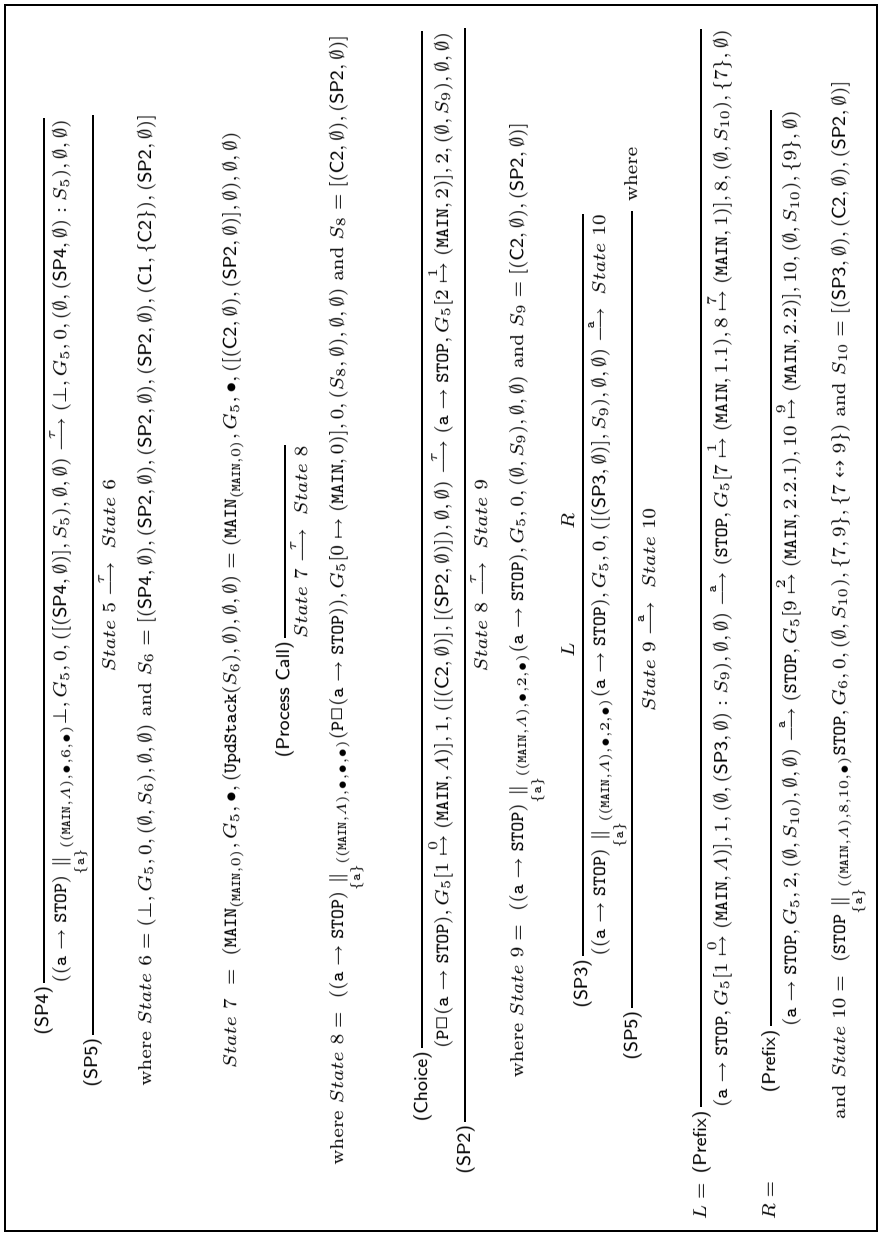


Fig. 5. An example of computation (step by step) with Algorithm 1 (cont.)

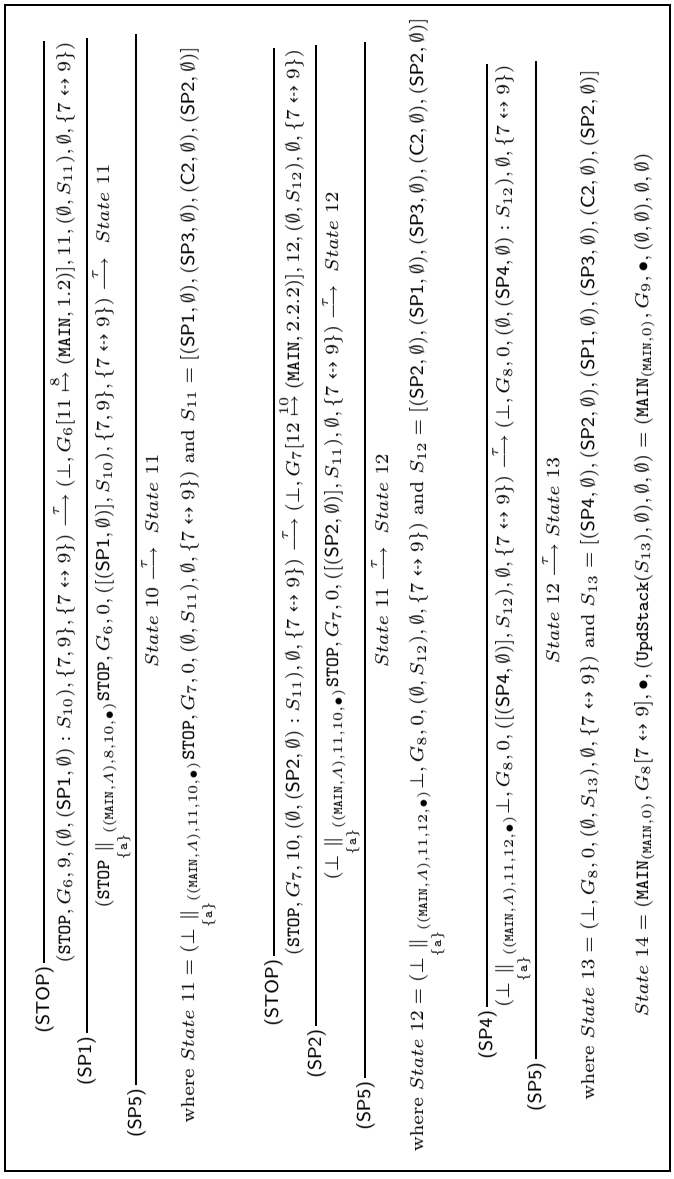


Fig. 5. An example of computation (step by step) with Algorithm 1 (cont.)

Example 4. Consider the following CSP specification where each literal has been labelled (they are underlined) with their associated specification position.

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

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

Following Algorithm 1, we use the initial state $(\mathtt{MAIN}_{(\mathtt{MAIN},0)}, \emptyset, \bullet, (\emptyset,\emptyset), \emptyset, \emptyset)$ to execute the semantics and get the computation of Fig. 7. This computation produces as a side effect the CSCFG shown in Fig. 6 for this specification. In this CSCFG, there is a loop edge between $(\mathtt{P}, 2)$ and $(\mathtt{MAIN}, 2)$. Note that the loop edge avoids infinite unfolding of the infinite process P, thus ensuring that the CSCFG is finite. Loop edges are introduced by the semantics whenever the context is repeated. In Fig. 7, when process P is called a second time, rule (Process call) unfolds P, its right-hand side is marked as a loop and a loop edge between nodes 7 and 2 is added to the graph. In *State* 4, the looped process is in parallel with a process waiting to synchronize with it. In order to perform the synchronization, the loop is unlabelled (*State* 5) by rule (SP4). Later, it is labelled again by rule (Process Call) when the loop is repeated (*State* 8 in Fig. 7 (cont.)). Finally, rule (SP4) detects that the left branch of the parallelism is deadlocked and the parallelism is marked as a loop (*State* 9), thus finishing the computation.

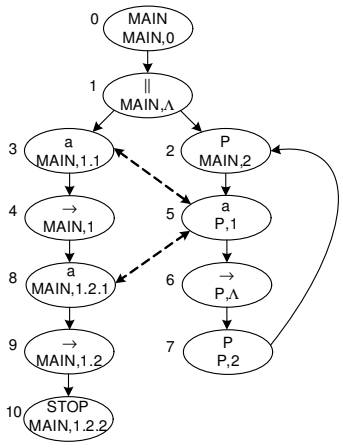


Fig. 6. CSCFG associated with the CSP specification in Example 4

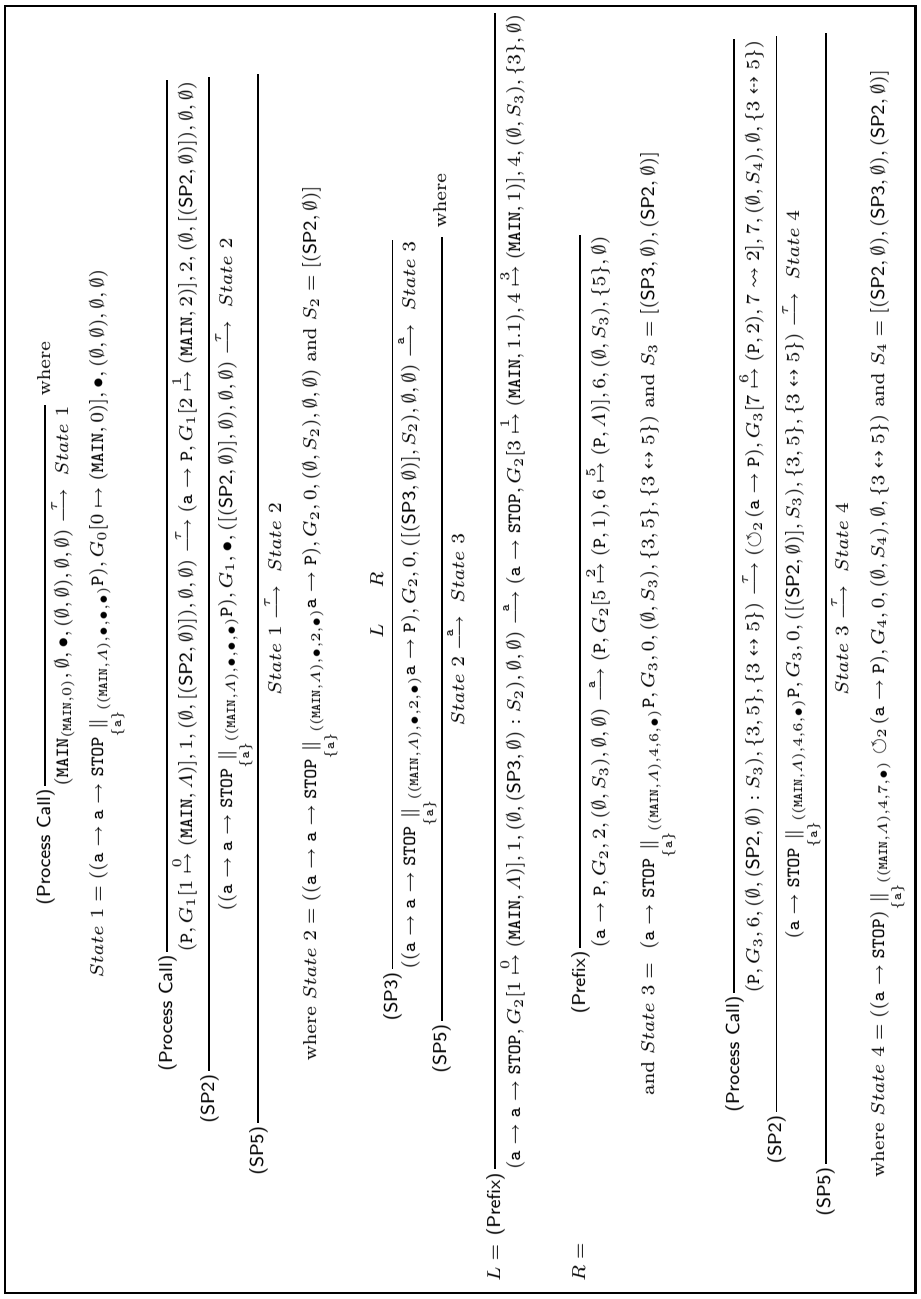


Fig. 7. Computation of the specification in Example 4 with Algorithm 1

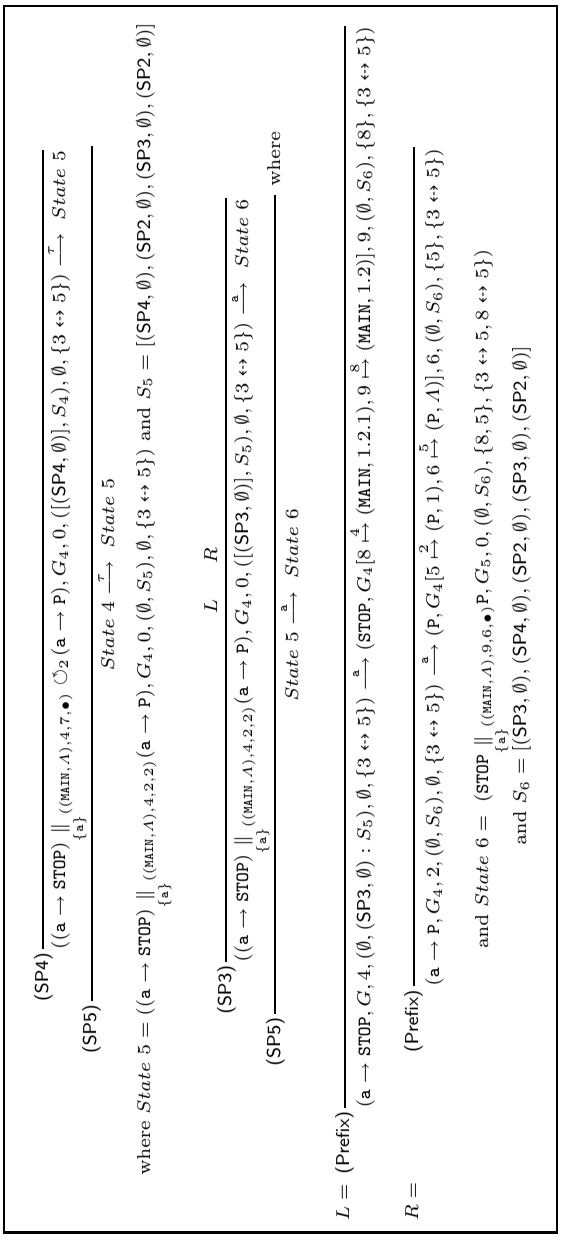


Fig. 7. Computation of the specification in Example 4 with Algorithm 1 (cont.)

5 Correctness

In this section we state the correctness of the proposed algorithm by showing that (i) the graph produced by the algorithm for a CSP specification $\mathcal{S}$ is the CSCFG

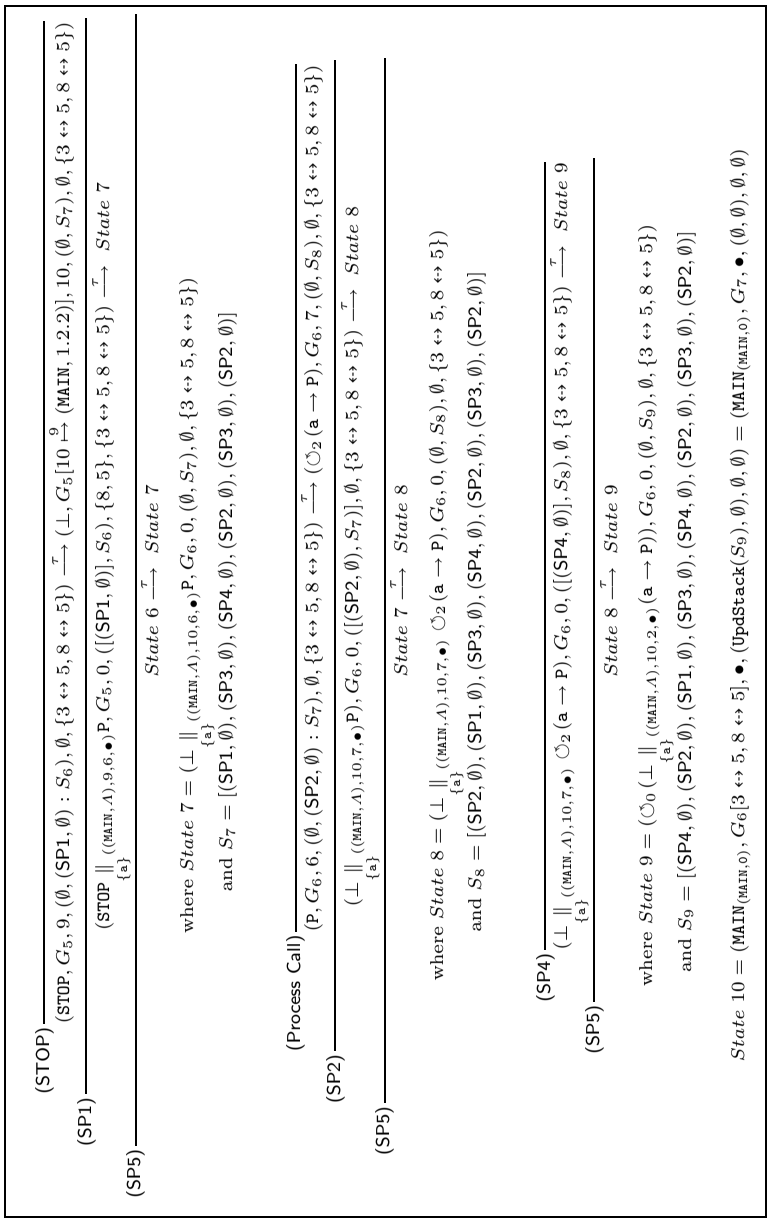


Fig. 7. Computation of specification in Example 4 with Algorithm 1 (cont.)

of $\mathcal{S}$; and (ii) the algorithm terminates, even if non-terminating computations exist for the specification $\mathcal{S}$. In order to prove these theorems, we need some preliminary definitions and lemmas.

Definition 6. *(Rewriting Step, Derivation) Given a state s of the instrumented semantics, a* rewriting step *for s ($s \overset{\Theta}{\rightsquigarrow} s'$) is the application of a rule of the semantics $\dfrac{\Theta}{s \xrightarrow{e} s'}$ with the occurrence of an event $e \in \Sigma^\tau$ and where Θ is a (possibly empty) set of rewriting steps. Given a state s_0, we say that the sequence $s_0 \overset{\Theta_0}{\rightsquigarrow} \dots \overset{\Theta_n}{\rightsquigarrow} s_{n+1}$, $n \geq 0$, is a* derivation *of s_0 iff $\forall\ i, 0 \leq i \leq n, s_i \overset{\Theta_i}{\rightsquigarrow} s_{i+1}$ is a rewriting step. We say that the derivation is* complete *iff there is no possible rewriting step for s_{n+1}. We say that two derivations $\mathcal{D}$, $\mathcal{D}'$ are* equivalent *(denoted $\mathcal{D} \equiv \mathcal{D}'$) iff all specification positions in the control of a rewriting step of $\mathcal{D}$ also appear in a rewriting step of $\mathcal{D}'$ and viceversa.*

The following lemma ensures that all possible derivations of $\mathcal{S}$ are explored by Algorithm 1.

Lemma 1. *Let $\mathcal{S}$ be a CSP specification and $\mathcal{D}$ a complete derivation of $\mathcal{S}$ performed with the standard semantics. Then, Algorithm 1 performs a derivation $\mathcal{D}'$ such that $\mathcal{D} \equiv \mathcal{D}'$.*

Proof. We prove first that the algorithm executes the instrumented semantics with a collection of initial states that explores all possible derivations. We prove this showing that every non-deterministic application of a rule is stored in the stack with all possible rules that can be applied; then, Algorithm 1 restarts the semantics with a new state that forces the semantics to explore a new derivation. This is done until all possible derivations have been explored.

Firstly, the standard semantics is deterministic except for two rules: (i) choice: the choice rules are evaluated until one branch is selected; and (ii) synchronized parallelism: the branches of the parallelism can be executed in any order.

In the case of choices, it is easy to see that the only applicable rule in the instrumented semantics is (Choice). Let us assume that we evaluate this rule with a pair of stacks (S, S_0). There are two possibilities in this rule: If S is empty, this rule puts in the control the left branch, and $[(\mathsf{C1}, \{\mathsf{C2}\})]$ is added to S_0, meaning that the left branch of the choice is executed and the right branch is pending. Therefore, we can ensure that the left branch is always explored because the algorithm evaluates the semantics with an initially empty stack. If the last element of S is either $(\mathsf{C1}, \{\mathsf{C2}\})$ or $(\mathsf{C2}, \emptyset)$, the semantics evaluates the first (resp. second) branch and deletes this element from S, and adds it to S_0.

We know that none of the other rules changes the stacks except (Synchronized Parallelism), and they both ((Synchronized Parallelism) and (Choice)) do it in the same manner. Therefore, we only have to ensure that the algorithm takes the stack S_0, selects another possibility (e.g., if C1 was selected in the previous evaluation, then C2 is selected in the next evaluation, i.e., if the head of the stack is $(\mathsf{C1}, \{\mathsf{C2}\})$ it is changed to $(\mathsf{C2}, \emptyset)$), puts it in the new initial state as the stack S, and the other stack is initialised for the next computation. This is exactly what the algorithm does by using function `UpdStack`.

In the case of synchronized parallelism, the semantics does exactly the same, but this case is a bit more complex because there are five different rules than can

be applied. In the standard semantics, non-determinism comes from the fact that both (Synchronized Parallelism 1) and (Synchronized Parallelism 2) can be executed with the same state. If this happens, the instrumented semantics executes one rule first and then the other, and all the way around in the next evaluation. When a parallelism operator is in the control and the stack is empty, rule (Synchronized Parallelism 5) is executed. This rule uses function `AppRules` to determine what rules could be applied. If non-determinism exists in the standard semantics, it also exists in the instrumented semantics, because the control of both semantics is the same except for the following cases:

`STOP` Rule (STOP) of the instrumented semantics 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 $\perp$ in the control. Then, the (sub)computation is stopped because no rule is applicable for $\perp$. Therefore, when the control in the derivation is STOP, the instrumented semantics performs one additional rewriting step with rule (STOP). Therefore, no additional non-determinism appears in the instrumented semantics due to (STOP).

$\perp$ This symbol only appears in the instrumented semantics. If it is in the control, the computation terminates because no rule can be applied. Therefore, no additional non-determinism appears in the instrumented semantics due to $\perp$.

$\circlearrowleft$ This symbol is introduced in the computation by (Process Call) or (Synchronized Parallelism 1, 2 and 3). Once it is introduced, there are two possibilities: (i) it cannot be removed by any rule, thus this case is analogous to the previous one; or (ii) it is removed by (Synchronized Parallelism 4) because the $\circlearrowleft$ is the label of a branch of a parallelism operator. In this case, the control remains the same as in the standard semantics, and hence, no additional non-determinism appears.

After (Synchronized Parallelism 5) has been executed, we have all possible applicable rules in the stack S, and S_0 remains unchanged. Then, the semantics executes the first rule, deletes it from S, and adds it to S_0. Therefore, the same mechanism used for choices is valid for parallelisms, and thus all branches of choices and parallelisms are explored.

Now, we have to prove that any possible (non-deterministic) derivation of `MAIN` with the standard semantics is also performed by the instrumented semantics as defined by Algorithm 1. We proof this lemma by induction on the length of the derivation $\mathcal{D}$.

In the base case, the initial state for the instrumented semantics induced by Algorithm 1 is in all cases $(\texttt{MAIN}_{(\texttt{MAIN},0)}, G, \bullet, (S, \emptyset), \emptyset, \emptyset)$ where $S = \emptyset$ in the first execution and $S \neq \emptyset$ in the other executions. Therefore, both semantics can only perform (Process Call) with an event τ. Hence, in the base case, both derivations are equivalent. We assume as the induction hypothesis, that both derivations

are equivalent after n steps of the standard semantics, and we prove that they are also equivalent in the step $n+1$.

The most interesting cases are those in which the event is an external event. All possibilities are the following:

- (STOP) In this case, both derivations finish the computation. The instrumented semantics performs one step more with the application of rule (STOP) (see the first item in the previous description).
- (Process Call) and (Prefixing) In these cases, both derivations apply the same rule and the control is the same in both cases.
- (Internal Choice 1 and 2) In these cases, the control becomes the left (resp. right) branch. They are analogous to the (Choice) rule of the instrumented semantics because both branches will be explored in different derivations as proved before.
- (External Choice 1,2,3 and 4) With (External Choice 1 and 2) only τ events can be performed several times to evolve the branches of the choice. In every step the final control has the same specification position of the choice operator. Finally, one step is applied with (External Choice 3 or 4). Then, the set of rewriting steps performed with external choice are of the form:

$$\frac{P_0 \xrightarrow{\tau} P_1}{(P_0 \,\Box\, Q) \xrightarrow{\tau} (P_1 \,\Box\, Q)} \cdots \frac{P_n \xrightarrow{e} P_{n+1}}{(P_n \,\Box\, Q) \xrightarrow{e} P_{n+1}}$$

  We can assume that (External Choice 1) is applied several times and finally (External Choice 3). This assumption is valid because (External Choice 2) is completely analogous to (External Choice 1); (External Choice 3) is completely analogous to (External Choice 4); and all combinations are going to be executed by the semantics as proved before. Then, we have an equivalent set of rewriting steps with the instrumented semantics:

$$\frac{}{(P_0 \,\Box\, Q) \xrightarrow{\tau} P_0}, \frac{}{P_0 \xrightarrow{\tau} P_1} \cdots \frac{}{P_n \xrightarrow{\tau} P_{n+1}}$$

  Clearly, in both sequences, the specification positions of the control are the same.
- (Synchronized Parallelism 1 and 2) Both rules can be applied interwound in the standard semantics. As it has been already demonstrated, we know that the same combination of rules will be applied by the instrumented semantics according to the algorithm use of the stack. The only difference is that the instrumented semantics performs an additional step with (Synchronized Parallelism 5), but this rule keeps the parallelism operator in the control; thus the specification position is the same and the claim holds.
- (Synchronized Parallelism 3) If this rule is applied in the standard semantics, in the instrumented semantics we apply (Synchronized Parallelism 5) and then (Synchronized Parallelism 3). The specification positions of the control do not change.

Lemma 2. *Let $\mathcal{S}$ be a CSP specification, and $\mathcal{D} = s_0 \overset{\Theta_0}{\rightsquigarrow} \ldots \overset{\Theta_n}{\rightsquigarrow} s_{n+1}$ a derivation of $\mathcal{S}$ performed with the instrumented semantics. Then, for each rewriting step*

$s_i \overset{\Theta_i}{\rightsquigarrow} s_{i+1}$, $0 \leq i < n$, *with* $s_i = (P_\alpha, G, m, (S, S_0), \Delta, \zeta)$, *and* $s_{i+1} = (Q, G', n, (S', S_0'), \Delta', \zeta')$*; we have that* $n \overset{m}{\mapsto} \alpha \in G'$.

Proof. The lemma trivially holds for all rules of the semantics. The only interesting case is synchronized parallelism. In the case of (Synchronized Parallelism 1, 2 and 3), function `InitBranch` inserts $n \overset{m}{\mapsto} \alpha$ into G', the first time it is evaluated. In the case of (Synchronized Parallelism 4), function `LoopCheck` returns another synchronized parallelism or a $\circlearrowleft$ only if one of the processes has been marked as a loop. This only happens if a process call has been unfolded; and in turn, this only happens if (Synchronized Parallelism 1, 2 or 3) has been performed. The other possibility is that function `LoopCheck` returns a $\perp$. In this case, $\perp$ cannot be further unfolded because no rule is applicable. Then, it must be the control of the state s_{n+1} and hence it is not required that $n \overset{m}{\mapsto} \alpha \in G'$. Finally, (Synchronized Parallelism 5) starts a subderivation with a parallelism operator in the control and a non-empty stack. Therefore, another of the previous rules must be applied after it, and thus, the claim follows.

Lemma 3. *Let* $\mathcal{S}$ *be a CSP specification, and* $G = (N, E_c, E_l, E_s)$ *the graph produced for* $\mathcal{S}$ *by Algorithm 1. Then, for each two nodes* $n, n' \in N$, $(n \mapsto n') \in E_c$ *iff the control can pass from* $l(n)$ *to* $l(n')$ *and* $\nexists n''$. $(n \mapsto n'') \in E_c$ *and* $(n'' \mapsto n') \in E_c$.

Proof. The fact that $\nexists n''$. $(n \mapsto n'') \in E_c$ and $(n'' \mapsto n') \in E_c$ implies that the control can pass from n to n' directly, i.e., without a transitive relation. This condition is needed because the CSCFG only contains control-flow edges between those nodes where the control can pass from one to the other directly. Moreover, all the arcs in E_c are added to G by the instrumented semantics. Therefore, we only have to prove that in every derivation $\mathcal{D}$ of the semantics, for every new arc $(n \mapsto n')$ added to E_c, the control can pass from $l(n)$ to $l(n')$. We prove this lemma by induction on the length of the derivation $\mathcal{D}$. The base case starts with the initial state $(\texttt{MAIN}_{(\texttt{MAIN},0)}, \emptyset, \bullet, (\emptyset, \emptyset), \emptyset, \emptyset)$. Therefore the only rule applicable is (Process Call). This case is trivial because in the new arc $n \overset{m}{\mapsto} \alpha$, $l(m) = (\texttt{MAIN}, 0)$ and $l(n) = (\texttt{MAIN}, \Lambda)$. Hence, by item (i) of Definition 2 we have that the control can pass from $l(m)$ to $l(n)$. We assume as the induction hypothesis that the lemma holds in the i first rewriting steps of $\mathcal{D}$, and we prove that it also holds in the step $i+1$. In the rewriting step $i+1$, one of the following rules must be applied:

- (Process Call) This case is analogous to the base case, because in the new added arc $n \overset{m}{\mapsto} \alpha$, m must be the name of a process N, and $n = (\texttt{N}, \Lambda)$. Therefore, by item (i) of Definition 2 we have that the control can pass from $l(m)$ to $l(n)$.
- (Prefixing) Two new arcs are added to G. $n \overset{m}{\mapsto} \alpha$ and $o \overset{n}{\mapsto} \beta$. Trivially, the control can pass from $l(n)$ to $l(o)$ by item (iii) of Definition 2. Moreover, by Lemma 2 we have that a node with the specification position of P and parent o will be added to G in the next rewriting step. Therefore, the control can pass from $l(o)$ to $\mathcal{P}os(P)$ by item (iv) of Definition 2.

- (Choice) One of the branches P' is the new control. Therefore, by Lemma 2 we have that a node with the specification position of P' and parent n will be added to G in the next rewriting step. Thus, the control can pass from $l(n)$ to the next fresh reference by item (ii) of Definition 2.
- (Synchronized Parallelism 1 and 2) They are analogous to the case of the choice.
- (Synchronized Parallelism 3) In this case, two new nodes are added. Each of them corresponds to one branch and are exactly the same as in (Synchronized Parallelism 1 and 2).
- (Synchronized Parallelism 4) This rule does not add new nodes to the graph.
- (Synchronized Parallelism 5) This rule starts a subderivation by applying one of the other rules associated with synchronized parallelism, thus the claim follows by the induction hypothesis.

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

Proof. After every execution of the semantics, Algorithm 1 introduces in the graph G all the synchronizations in the set ζ. Therefore, we have to prove that at the end of the derivation $\mathcal{D}$, all the synchronizations are in ζ.

We prove this lemma by induction on the length of the derivation $\mathcal{D} = s_0 \overset{\Theta_0}{\rightsquigarrow} s_1 \overset{\Theta_1}{\rightsquigarrow} \dots \overset{\Theta_n}{\rightsquigarrow} s_{n+1}$. We can assume that the derivation starts with the initial state $(\texttt{MAIN}_{(\texttt{MAIN},0)}, \emptyset, \bullet, (\emptyset, \emptyset), \emptyset, \emptyset)$, 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 $(a \leftrightarrow a') \in E_s$ for each synchronization in $s_0 \overset{\Theta_0}{\rightsquigarrow} \dots \overset{\Theta_{i-1}}{\rightsquigarrow} s_i$ with $0 < i \leq n$ and prove that the lemma also holds for the next rewriting step $s_i \overset{\Theta_i}{\rightsquigarrow} s_{i+1}$.

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

$$\frac{Left \quad Right}{(P1 \parallel_X^{(\alpha, n_1, n_2, \Upsilon)} P2, G, m, (S' : (\mathsf{SP3}, rules), S_0), \Delta, \zeta) \xrightarrow{e} (P', G'', m, (S''', S_0''), \Delta_1 \cup \Delta_2, \zeta' \cup syncs)}$$

where

$$(G_1', \zeta_1, n_1') \;=\; \texttt{InitBranch}(G, \zeta, n_1, m, \alpha) \wedge e \in X$$

$$Left = (P1, G_1', n_1', (S', (\mathsf{SP3}, rules) : S_0), \Delta, \zeta_1) \xrightarrow{e} (P1', G_1'', n_1'', (S'', S_0'), \Delta_1, \zeta_1')$$

$$(G_2', \zeta_2, n_2') \;=\; \texttt{InitBranch}(G_1'', \zeta_1', n_2, m, \alpha)$$

$$Right = (P2, G_2', n_2', (S'', S_0'), \Delta, \zeta_2) \xrightarrow{e} (P2', G'', n_2'', (S''', S_0''), \Delta_2, \zeta')$$

$$sync = \{s_1 \leftrightarrow s_2 \mid s_1 \in \Delta_1 \wedge s_2 \in \Delta_2\} \wedge \forall\ (m \leftrightarrow n) \in sync\ .\ G''[m \leftrightarrow n]$$

$$P' = \begin{cases} \circlearrowleft_m\ (\texttt{Unloop}(P1' \underset{X}{\parallel}_{(\alpha, n_1'', n_2'', \bullet)} P2')) & \text{if } (sync \cup \zeta') = \zeta \\ P1' \underset{X}{\parallel}_{(\alpha, n_1'', n_2'', \bullet)} P2' & \text{otherwise} \end{cases}$$

Because (Prefixing) is the only rule that performs an a event without further conditions, we know that $P1$ must be a prefixing operator or a parallelism containing a prefixing operator whose prefix is a, i.e., we know that the rule applied in $Left$ 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 $P2$. Hence, two prefixing rules (one for $P1$ and one for $P2$) have been fired as a condition of this rule.

In addition, the new set ζ contains the synchronization set $\{s_1 \leftrightarrow s_2 \mid s_1 \in \Delta_1 \wedge s_2 \in \Delta_2\}$ where Δ_1 and Δ_2 are the sets of references to the events that must synchronize in $Left$ and $Right$, respectively.

Hence, we have to prove that all and only the events (a) that must synchronize in $Left$ are in Δ_1. 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 $Right$.

The only applicable rules in

$$(P1, G_1', n_1', (S' : (\mathsf{SP3}, rules), S_0), \Delta, \zeta) \xrightarrow{e} (P1', G_1'', n_1'', (S'', S_0'), \Delta_1, \zeta')$$

are:

- (Prefixing) In this case, the prefix a is added to Δ_1.
- (Synchronized Parallelism 1, 2 and 5) In these cases, the set Δ' is propagated down.
- (Synchronized Parallelism 3) In this case, the sets Δ_1 and Δ_2 are joined and propagated down.

Therefore, all the synchronized events are in the set Δ_1 and the claim follows.

Lemma 5. *Let $\mathcal{S}$ be a CSP specification and $G = (N, E_c, E_l, E_s)$ the CSCFG produced by Algorithm 1 for $\mathcal{S}$. Then, $(n_1 \rightsquigarrow n_2) \in E_l$ iff $l(n_1)$ and $l(n_2)$ are process calls that refer to the same process $M \in \mathcal{N}$ and $n_2 \in \mathcal{C}on(n_1)$.*

Proof. First, all edges in E_l are introduced in a derivation $\mathcal{D}$ of the instrumented semantics. Let $\mathcal{D} = s_0 \overset{\Theta_0}{\rightsquigarrow} \ldots \overset{\Theta_n}{\rightsquigarrow} s_{n+1}$ a derivation that introduced $(n_1 \rightsquigarrow n_2)$ into E_l. Then, this arc is necessarily introduced in a rewriting step where rule (Process Call) was applied, because this is the only rule that adds arcs to E_l. In rule (Process Call), arcs are added by means of function `LoopCheck`. An arc is added to E_l if and only if $\exists n_2 \in Path(0, n_1) \ \wedge\ n_2 \overset{t}{\mapsto} M \in E_c$, where n_1 is the reference of the current node added to N. Therefore, because function `LoopCheck` adds the arc $n_1 \rightsquigarrow n_2$, then $l(n_1) = l(n_2) = M$. Hence, we need to prove that $n_2 \in \mathcal{C}on(n_1)$.

First, by Lemma 3, if the control can pass (transitively) from n_2 to n_1, then we have in G a path of control edges $n_2 \mapsto^* n_1$. We can show that this path is loop-free by contradiction. Let us consider that the path is not loop-free. Then, $n_2 \mapsto^* n_3 \mapsto^* n_1$ with $l(n_3) = M$ and $n_1 \neq n_3$. The derivation $\mathcal{D}$ must be of the form:

$$\mathcal{D} = s_0 \overset{\Theta_0}{\rightsquigarrow} \dots s_i \overset{\Theta_i}{\rightsquigarrow} s_{i+1} \dots \overset{\Theta_n}{\rightsquigarrow} s_{n+1}, 0 < i \leq n$$

where the rewriting step $s_i \overset{\Theta_i}{\rightsquigarrow} s_{i+1}$ introduced n_3 in G. Clearly, n_3 is necessarily introduced in G by rule (Process Call) which is the only rule that adds a process call to the graph. Moreover, by the definition of `LoopCheck` and because $\exists n_2 \ . \ n_2 \overset{t}{\mapsto} M \in G \wedge \ n_2 \in Path(0, n_3)$, we know that the control of s_{i+1} is $\circlearrowleft_{n_2}(rhs(N))$. But this is a contradiction with the fact that $n_3 \mapsto^* n_1$ because no rule of the semantics adds a control-flow edge of the form $n_3 \mapsto$. In particular, once the control of s_{i+1} is labelled with $\circlearrowleft_{n_2}$, only the rule (Synchronized Parallelism 4) can remove the label of the control. This is done with function `LoopControl` in the third case of the definition. But in this case, the parallelism is marked as $P'_{\circlearrowleft} \underset{X}{\|}_{(\alpha, p_{\circlearrowleft}, q', p_{\circlearrowleft})} Q'$ where $p_{\circlearrowleft}$ is the label of the previous process call to M. Hence, $p_{\circlearrowleft} = n_2$; and thus, the next control edges added to G start from n_2, and not from n_3.

Theorem 1 (Correctness) *Let $\mathcal{S}$ be a CSP specification and G the graph produced for $\mathcal{S}$ by Algorithm 1. Then, G is the CSCFG associated with $\mathcal{S}$.*

Proof. In order to prove that G is a CSCFG, we need to prove that it satisfies the properties of Definition 5. Let us consider a CSCFG $G = (N, E_c, E_l, E_s)$.

Firstly, by Lemma 3, and because control-flow is a transitive relation, we know that for each rewriting step in a derivation of $\mathcal{S}$ the control can pass from `MAIN` to the positions added to N. Hence, $\forall \ n \in N. \ l(n) \in \mathcal{P}os(\mathcal{S})$ and $l(n)$ is executable in $\mathcal{S}$. In addition, we have that:

- by Lemma 3, for each two nodes $n, n' \in N$, $(n \mapsto n') \in E_c$ iff the control can pass from n to n'.
- by Lemma 5, $(n_1 \rightsquigarrow n_2) \in E_l$ iff $l(n_1)$ and $l(n_2)$ are (possibly different) process calls that refer to the same process $M \in \mathcal{N}$ and $n_2 \in \mathcal{C}on(n_1)$;
- by Lemma 4, there exists a synchronization edge $(a \leftrightarrow a')$ in G for each synchronization in a derivation $\mathcal{D}$ of $\mathcal{S}$ where a and a' are the nodes of the synchronized events. And, by Lemma 1 we know that all possible derivations of $\mathcal{S}$ are explored by Algorithm 1.

Moreover, we know that the only nodes in N are the nodes induced by E_c because all the nodes added to G are added by connecting the new node to the last added node (i.e., if the current reference is m and the new fresh reference is n, then the new node is always added as $G[n \overset{m}{\mapsto} \alpha]$). Hence, all nodes are related by control edges and thus the claim holds.

Theorem 2 (Termination) *Let $\mathcal{S}$ be a CSP specification. Then, the execution of Algorithm 1 with $\mathcal{S}$ terminates.*

Proof. In order to prove that the algorithm terminates we have to show that the stack never grows infinitely. For this purpose, we have to prove that all executions of the semantics terminate. This is sufficient because function `UpdStack`, which is the only one that also manipulates the stack, always either reduces its size or leaves it unchanged. So, as the stack is always increased by rule (Synchronized Parallelism 5) or by rule (Choice), we have to show that there is no derivation which fires these rules infinitely. We use a function over sets of rewriting steps which is defined as follows:

$$[\mathcal{R}] = \bigcup\{\ \{s \overset{\Theta}{\rightsquigarrow} s'\} \cup [\Theta] \mid s \overset{\Theta}{\rightsquigarrow} s' \in \mathcal{R}\}$$

Given a set $\mathcal{R}$ of rewriting steps, it returns $\mathcal{R}$ and all the rewriting steps included in the subderivations of $\mathcal{R}$.

In the following, we will consider derivations where the state is simplified and only the control is taken into account. In order to prove that there does not exist any infinite derivation, we consider the main derivation $\mathcal{D}$ of the semantics where the initial control is `MAIN`. If $\forall s_i \overset{\Theta_i}{\rightsquigarrow} s_{i+1} \in \mathcal{D}$ where $\nexists s \overset{\Theta}{\rightsquigarrow} s' \in [\{s_i \overset{\Theta_i}{\rightsquigarrow} s_{i+1}\}]$ such that $s = N$ and $s' = \circlearrowleft(rhs(N))$, then we know that the derivation $\mathcal{D}$ is finite because no infinite unfolding is possible (we know that no process is called twice) and the specification is finite. Hence, as the application of the rules of the semantics always reduces the size of the process in the control, it will eventually terminate with $\perp$.

The other case happens when the same process appears twice in a derivation. We can assume that, after a number of rewriting steps, we find the first occurrence of a rewriting step $s_i \overset{\Theta_i}{\rightsquigarrow} s_{i+1} \in \mathcal{D}$ where $\exists s \overset{\Theta}{\rightsquigarrow} s' \in [\{s_i \overset{\Theta_i}{\rightsquigarrow} s_{i+1}\}]$ such that $s = N$ and $s' = \circlearrowleft(rhs(N))$. When this happens, we know that N has been already unfolded in a previous rewriting step, and function `LoopCheck` introduces the loop s' through the rule (Process Call) which corresponds to $s \overset{\Theta}{\rightsquigarrow} s'$.

We have two possibilities: the first one happens when $s' = s_{i+1}$ which means that this is the last rewriting step of derivation $\mathcal{D}$ since there does not exist any rule for $\circlearrowleft(\_)$. In the other case, when $s' \neq s_{i+1}$, we have that rewriting step $s_i \overset{\Theta_i}{\rightsquigarrow} s_{i+1}$ corresponds to the application of rule (Synchronized Parallelism 1), (Synchronized Parallelism 2) or (Synchronized Parallelism 5), since no other rule can fire the rule (Process Call) into the associated Θ_i. Note that rule (Synchronized Parallelism 3) can not be applied here because event τ can not fire this rule. This means that s_i is a parallelism which has nested parallelisms in its branches and some of these branches has the process call N. Then we know that $\exists\ (s \underset{X}{\parallel} P) \overset{\Theta'}{\rightsquigarrow} (s' \underset{X}{\parallel} P) \in [\{s_i \overset{\Theta_i}{\rightsquigarrow} s_{i+1}\}]$ where $\Theta' = \{s \overset{\Theta}{\rightsquigarrow} s'\}$.[3] Now, process P could be of one of these kinds:

[3] Of course, s could be on the right branch of the parallelism, but we only consider this case since the other one is analogous.

- $\bot$: In this case, there is a rewriting step $s_j \overset{\Theta_j}{\rightsquigarrow} s_{j+1} \in \mathcal{D}$ with $j > i$ such that $(s' \underset{X}{\|} \bot) \overset{\Theta''}{\rightsquigarrow} \circlearrowleft (s' \underset{X}{\|} \bot) \in [\{s_j \overset{\Theta_j}{\rightsquigarrow} s_{j+1}\}]$ by application of rule (Synchronized Parallelism 4). Then, if $s_{j+1} = \circlearrowleft (s' \underset{X}{\|} \bot)$ then the computation terminates. Else, s_{j+1} is a parallelism and it terminates by induction.
- $\circlearrowleft (Q')$: In this case, there is a rewriting step $s_j \overset{\Theta_j}{\rightsquigarrow} s_{j+1} \in \mathcal{D}$ with $j > i$ such that $(s' \underset{X}{\|} \circlearrowleft (Q')) \overset{\Theta''}{\rightsquigarrow} \circlearrowleft (s' \underset{X}{\|} Q') \in [\{s_j \overset{\Theta_j}{\rightsquigarrow} s_{j+1}\}]$ by application of rule (Synchronized Parallelism 4). Then, if $s_{j+1} = \circlearrowleft (s' \underset{X}{\|} Q')$ then the computation terminates. Else, s_{j+1} is a parallelism and it terminates by induction.
- STOP: Then, there is some rewriting step $s_j \overset{\Theta_j}{\rightsquigarrow} s_{j+1} \in \mathcal{D}$ with $j > i$ such that $(s' \underset{X}{\|} \texttt{STOP}) \overset{\Theta''}{\rightsquigarrow} (s' \underset{X}{\|} \bot) \in [\{s_j \overset{\Theta_j}{\rightsquigarrow} s_{j+1}\}]$, and it terminates by case $\bot$.
- $a \to Q$: Then, there are two possibilities. If $a \notin X$ then there is some rewriting step $s_j \overset{\Theta_j}{\rightsquigarrow} s_{j+1} \in \mathcal{D}$ with $j > i$ such that $(s' \underset{X}{\|} (a \to Q)) \overset{\Theta''}{\rightsquigarrow} (s' \underset{X}{\|} Q) \in [\{s_j \overset{\Theta_j}{\rightsquigarrow} s_{j+1}\}]$, then it terminates by induction. Else, when $a \in X$, there is a rewriting step $s_j \overset{\Theta_j}{\rightsquigarrow} s_{j+1} \in \mathcal{D}$ with $j > i$ such that $(s' \underset{X}{\|} (a \to Q)) \overset{\Theta''}{\rightsquigarrow} (rhs(N) \underset{X}{\|} a \to Q) \in [\{s_j \overset{\Theta_j}{\rightsquigarrow} s_{j+1}\}]$, where parallelism's Υ is equal to the label of the loop. Then, we have again two options. The first one is that some synchronization is drawn before to have N again in the left branch of the parallelism. Then, we have a rewriting step $s_k \overset{\Theta_k}{\rightsquigarrow} s_{k+1} \in \mathcal{D}$ with $k > j$ such that $(s'' \underset{X}{\|} a \to Q) \overset{\Theta'''}{\rightsquigarrow} (s''' \underset{X}{\|} Q) \in [\{s_k \overset{\Theta_k}{\rightsquigarrow} s_{k+1}\}]$, and Υ is put to $\bullet$ if the synchronization is not included in ζ yet. This case terminates, by induction hypothesis. Otherwise, if the synchronization was in ζ, then $(s'' \underset{X}{\|} a \to Q) \overset{\Theta'''}{\rightsquigarrow} \circlearrowleft (s''' \underset{X}{\|} Q) \in [\{s_k \overset{\Theta_k}{\rightsquigarrow} s_{k+1}\}]$ by rule (Synchronized Parallelism 3). If $s_{k+1} = \circlearrowleft (s''' \underset{X}{\|} Q)$, the derivation has terminated, else termination is proved by induction. The second case is when none synchronization is drawn before to have N again into the left branch. In this case, we have that $s_k \overset{\Theta_k}{\rightsquigarrow} s_{k+1} \in \mathcal{D}$ with $k > j$ such that $(s' \underset{X}{\|} a \to Q) \overset{\Theta'''}{\rightsquigarrow} \circlearrowleft (s' \underset{X}{\|} \bot) \in [\{s_k \overset{\Theta_k}{\rightsquigarrow} s_{k+1}\}]$ by rule (Synchronized Parallelism 4). If $s_{k+1} = \circlearrowleft (s' \underset{X}{\|} \bot)$, the derivation has terminated, else, termination is proved by induction.
- $Q_1 \,\square\, Q_2$: In this case, one of the branches is selected, and independently of which one is followed, the computation terminates by induction. Then, there is some rewriting step $s_j \overset{\Theta_j}{\rightsquigarrow} s_{j+1} \in \mathcal{D}$ with $j > i$ such that $(s' \underset{X}{\|} (Q_1 \,\square\, Q_2)) \overset{\Theta''}{\rightsquigarrow} (s' \underset{X}{\|} Q_1) \in [\{s_j \overset{\Theta_j}{\rightsquigarrow} s_{j+1}\}]$ or $(s' \underset{X}{\|} (Q_1 \,\square\, Q_2)) \overset{\Theta''}{\rightsquigarrow} (s' \underset{X}{\|} Q_2) \in [\{s_j \overset{\Theta_j}{\rightsquigarrow} s_{j+1}\}]$.

– $Q_1 \underset{Y}{\parallel} Q_2$: Using the induction hypothesis, a parallelism always terminates, so we have to consider that in this case Q will be rewritten either to $\perp$ or to $\circlearrowleft(Q'_1 \underset{Y}{\parallel} Q'_2)$ and thus, the computation finishes.

Therefore, the claim holds.

6 Conclusions

This work introduces an algorithm to build the CSCFG associated with a CSP specification. The algorithm uses an instrumentation of the standard CSP's operational semantics to explore all possible computations of a specification. The semantics is deterministic because the rule applied in every step is predetermined by the initial state and the information in the stack. Therefore, the algorithm can execute the semantics several times to iteratively explore all computations and hence, generate the whole CSCFG. The CSCFG is generated even for non-terminating specifications due to the use of a loop detection mechanism controlled by the semantics. This semantics is an interesting result because it can serve as a reference mark to prove properties such as completeness of static analyses based on the CSCFG. The way in which the semantics has been instrumented can be used for other similar purposes with slight modifications. For instance, the same design could be used to generate other graph representations of a computation such as Petri nets [10].

On the practical side, we have implemented a tool called *SOC* [8] which is able to automatically generate the CSCFG of a CSP specification. The CSCFG is later used for debugging and program simplification. *SOC* has been integrated into the most extended CSP animator and model-checker ProB [2, 6], that shows the maturity and usefulness of this tool and of CSCFGs. The last release of *SOC* implements the algorithm described in this paper. However, in the implementation the algorithm is much more complex because it contains some improvements that significantly speed up the CSCFG construction. The most important improvement is to avoid repeated computations. This is done by: (i) state memorization: once a state already explored is reached the algorithm stops this computation and starts with another one; and (ii) skipping already performed computations: computations do not start from `MAIN`, they start from the next non-deterministic state in the execution (this is provided by the information of the stack).

The implementation, source code and several examples are publicly available at: `http://users.dsic.upv.es/~jsilva/soc/`

References

1. Brassel, B., Hanus, M., Huch, F., Vidal, G.: A Semantics for Tracing Declarative Multi-paradigm Programs. In: Moggi, E., Warren, D.S. (eds.) 6th ACM SIGPLAN Int'l Conf. on Principles and Practice of Declarative Programming (PPDP'04), pp. 179–190. ACM, New York, NY, USA (2004)

2. Butler, M., Leuschel, M.: Combining CSP and B for Specification and Property Verification. In: Fitzgerald, J., Hayes, I.J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, pp. 221–236. Springer, Heildeberg (2005)
3. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Upper Saddle River, NJ, USA (1985)
4. Kavi, K.M., Sheldon, F.T., Shirazi, B., Hurson, A.R.: Reliability Analysis of CSP Specifications using Petri Nets and Markov Processes. In: 28th Annual Hawaii Int'l Conf. on System Sciences (HICSS'95), vol. 2 (Software Technology), pp. 516–524. IEEE Computer Society, Washington, DC, USA (1995)
5. Ladkin, P., Simons, B.: Static Deadlock Analysis for CSP-Type Communications. Responsive Computer Systems (Chapter 5), Kluwer Academic Publishers (1995)
6. Leuschel, M., Butler, M.: ProB: an Automated Analysis Toolset for the B Method. Journal of Software Tools for Technology Transfer. 10(2), 185–203 (2008)
7. Leuschel, M., Llorens, M., Oliver, J., Silva, J., Tamarit, S.: Static Slicing of CSP Specifications. In: Hanus, M. (ed.) 18th Int'l Symp. on Logic-Based Program Synthesis and Transformation (LOPSTR'08), pp. 141–150. Technical report, DSIC-II/09/08, Universidad Politécnica de Valencia (July 2008)
8. Leuschel, M., Llorens, M., Oliver, J., Silva, J., Tamarit, S.: SOC: a Slicer for CSP Specifications. In: Puebla, G., Vidal, G. (eds.) 2009 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM'09), pp. 165–168. ACM, New York, NY, USA (2009)
9. Leuschel, M., Llorens, M., Oliver, J., Silva, J., Tamarit, S.: The MEB and CEB Static Analysis for CSP Specifications. In: Hanus, M. (ed.) LOPSTR 2008, Revised Selected Papers. LNCS, vol. 5438, pp. 103–118. Springer, Heildeberg (2009)
10. Llorens, M., Oliver, J., Silva, J., Tamarit, S.: Transforming Communicating Sequential Processes to Petri Nets. In: Topping, B.H.V., Adam, J.M., Pallarés, F.J., Bru, R., Romero, M.L. (eds.) Seventh Int'l Conf. on Engineering Computational Technology (ICECT'10). Civil-Comp Press, Stirlingshire, Scotland (to appear 2010)
11. Roscoe, A.W., Gardiner, P.H.B., Goldsmith, M., Hulance, J.R., Jackson, D.M., Scattergood, J.B.: Hierarchical Compression for Model-Checking CSP or How to Check 10^{20} Dining Philosophers for Deadlock. In: Brinksma, E., Cleaveland, R., Larsen, K.G., Margaria, T., Steffen, B. (eds.) TACAS 1995. LNCS, vol. 1019, pp. 133–152. Springer, London (1995)
12. Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice Hall, Upper Saddle River, NJ, USA (2005)