# Graph Generation to Statically Represent CSP Processes*

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, 13] 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 [12], 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

* 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 ACOMP/2010/042, and by the *Universidad Politécnica de Valencia* (Program PAID-06-08). Salvador Tamarit was partially supported by the Spanish MICINN under FPI grant BES-2009-015019.

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 [13]) 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, 13]. 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 $P = M \mid a \rightarrow P \mid P \sqcap Q \mid P \,\square\, Q \mid P \underset{X \subseteq \Sigma}{||} Q \mid STOP$. 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** ($P \,\square\, Q$) 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 [13]. 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 $\Sigma$ are visible from the external environment, and can only happen with its co-operation (e.g., actions of the user). Event $\tau$ 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.

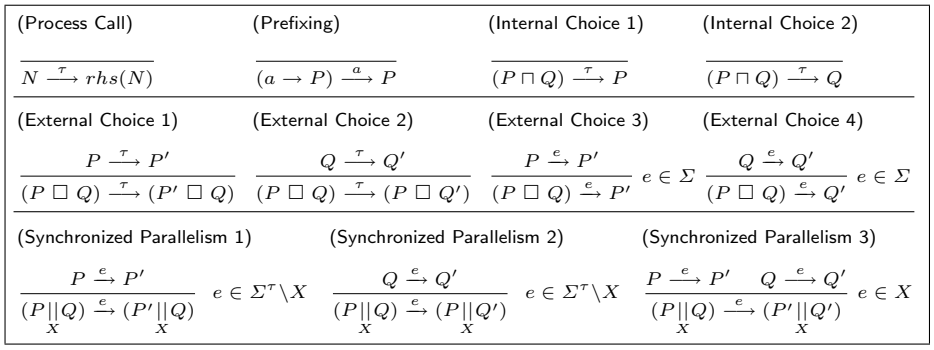



**Fig. 1.** CSP's operational semantics

## 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* $\Lambda$ *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:*

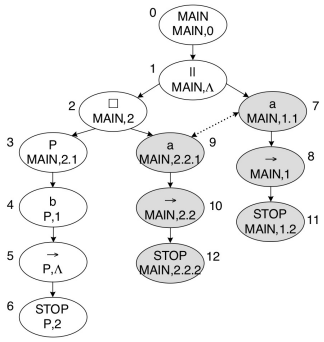
$$\texttt{AddSpPos}(P,(N,w)) = \begin{cases} P_{(N,w)} & \textit{if } \; P \in \mathcal{N} \\ STOP_{(N,w)} & \textit{if } \; P = STOP \\ a_{(N,w.1)} \rightarrow_{(N,w)} \texttt{AddSpPos}(Q,(N,w.2)) & \textit{if } \; P = a \rightarrow Q \\ \texttt{AddSpPos}(Q,(N,w.1))\ op_{(N,w)}\ \texttt{AddSpPos}(R,(N,w.2))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

*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.

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

(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 $\alpha, \beta$ in $\mathcal{S}$, we say that the* control *of $\alpha$* can pass *to $\beta$ iff*

*i)* $\alpha = N \ \wedge \ \beta = \mathit{first}((N, \Lambda))$ with $N = \mathit{rhs}(N) \in \mathcal{S}$
*ii)* $\alpha \in \{\sqcap, \Box, ||\} \ \wedge \ \beta \in \{\mathit{first}(\alpha.1), \mathit{first}(\alpha.2)\}$
*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 $\alpha$ is* executable *in $\mathcal{S}$ iff the control can pass from the initial state (i.e.,* `MAIN`*) to $\alpha$.*

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 $n_1$, $m \in N$, $\mathit{Path}(n_1, m)$, 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 $l(n) = l(m) \in \mathit{Names}$. 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 $(M, \Lambda)$ 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;[1] 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

[1] We refer the reader to [10] where an example with loop edges is discussed.

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 [13] 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., $P_\alpha$). 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, $\Delta$ is a set of references to nodes used to draw synchronizations in $G$ and $\zeta$ 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[2] $n$ 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 $\alpha$ and its

[2] 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 $\alpha$).

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.

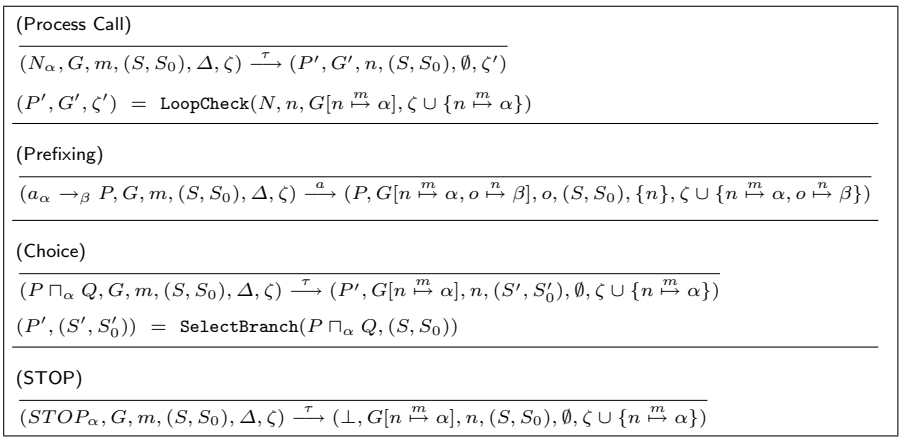



**Fig. 3.** An instrumented operational semantics that generates the CSCFG

(Process Call) The called process $N$ is unfolded, node $n$ (a fresh reference) is added to the graphs $G$ and $\zeta$ with specification position $\alpha$ 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 $\Delta$ 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, a loop edge between nodes $n$ and $s$ is added to the graph $G$ and to $\zeta$; and the right-hand

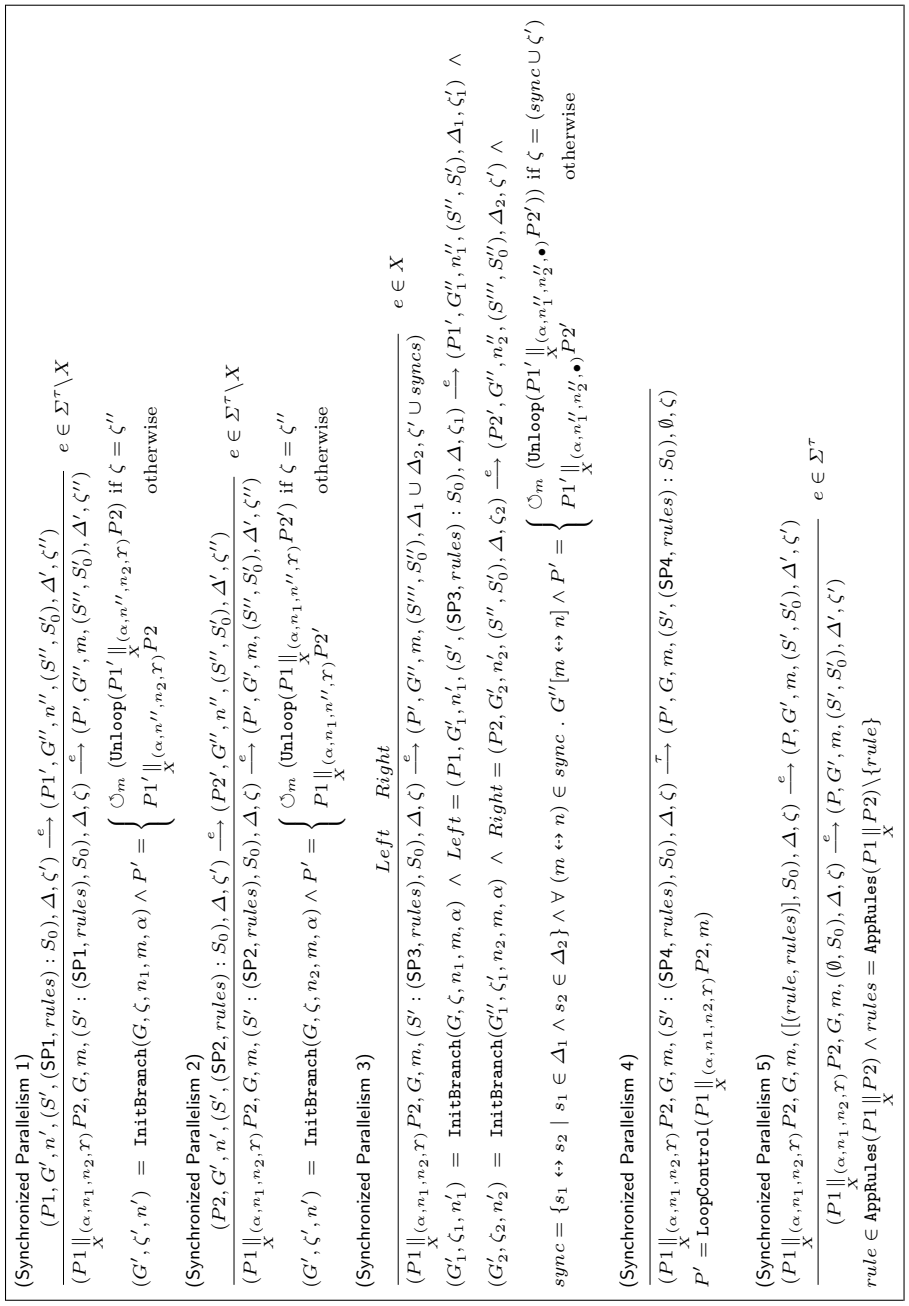



**Fig. 3.** An instrumented operational semantics that generates the CSCFG (cont.)

side of the called process is labelled with a special symbol $\circlearrowleft_s$. This label is later used by rule (Synchronized Parallelism 4) to decide whether the process must be stopped. The loop symbol $\circlearrowleft$ is labelled with the position $s$ of the process call of the loop. This is 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 $\zeta$. In the new state, $o$ becomes the current reference. The new control is $P$. The set $\Delta$ 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 $\alpha$ and has parent $m$. In the new state, $n$ becomes the current reference. No event can synchronize in this rule, thus $\Delta$ 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 $\alpha$ 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 $\Upsilon$ 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 $\Delta'$ and $\zeta''$ are passed down unchanged so that another rule can use them if necessary. In the case that $\zeta$ is equal to $\zeta''$, meaning that nothing has change in this derivation, 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`). 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 and into $\zeta$ 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, $\Upsilon$ 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 $P1$ ($\Delta_1$) and $P2$ ($\Delta_2$) are mutually synchronized and added to both $G''$ and $\zeta'$.

(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[3] 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,

[3] Only once because it will be labelled again by rule (Process Call) when the loop is repeated. In [10], we present an example with loops where this situation happens.

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`. 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):

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

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

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 $\Upsilon$ is put to $\bullet$. (ii) Either it is a loop that has been unfolded without drawing any synchronization (this is known because $\Upsilon$ 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, $\Upsilon$ is put to $\bullet$. (iii) If we are not in a loop, then we allow the parallelism to proceed by unlabelling the looped branch. When none of the branches has been labelled as a loop, $\bot$ 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.

$$\texttt{FstEvs}(P) =$$

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

(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 performs two iterations (one for each computation) to generate the final CSCFG. Figure 2(b) shows the CSCFG generated where white nodes were produced in the first iteration; and grey nodes were produced in the second iteration. For the interested reader, in [10] all computation steps executed by Algorithm 1 to obtain the CSCFG associated with the specification in Example 1 are explained in detail.

## 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 of $\mathcal{S}$; and (ii) the algorithm terminates, even if non-terminating computations exist for the specification $\mathcal{S}$.

**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 $\mathcal{S}$.*

This theorem can be proved by showing first that each step performed with the standard semantics has an associated step in the instrumented semantics; and that the specification position of the expression in the control is added to the CSCFG as a new node which is properly inserted into the CSCFG. This can be proved by induction on the length of a derivation in the standard semantics. Then, it must be proved that the algorithm performs all possible computations. This can be done by showing that every non-deterministic step of the semantics

is recorded in the stack with all possible rules that can be applied; and the algorithm traverses the stack until all possibilities have been evaluated. The interesting case of the proof happens when the computation is infinite. In this case, the context of a process call must be repeated because the number of process calls is finite by definition. Therefore, in this case the proof must show that functions `LoopCheck` and `LoopControl` correctly finish the computation. The proof of this theorem can be found in [10].

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

The proof of this theorem must ensure that all derivations of the instrumented semantics are finite, and that the number of derivations fired by the algorithm is also finite. This can be proved by showing that the stacks never grow infinitely, and they will eventually become empty after all computations have been explored. The proof of this theorem can be found in [10].

## 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 [11].

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.: A Semantics to Generate the Context-sensitive Synchronized Control-Flow Graph (extended). Technical report DSIC, Universidad Politécnica de Valencia. Accessible via `http://www.dsic.upv.es/~jsilva`, Valencia, Spain, June 2010.
11. 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)
12. 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)
13. Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice Hall, Upper Saddle River, NJ, USA (2005)