Document downloaded from:

http://hdl.handle.net/10251/34942

This paper must be cited as:

Leuschel ., M.; Llorens Agost, ML.; Oliver Villarroya, J.; Silva Galiana, JF.; Tamarit Muñoz, S. (2012). Static slicing of explicitly synchronized languages. Information and Computation. 214:10-46. doi:10.1016/j.ic.2012.02.005.

The final publication is available at

http://dx.doi.org/10.1016/j.ic.2012.02.005

Copyright Elsevier

Static Slicing of Explicitly Synchronized Languages☆

Michael Leuschel[a], Marisa Llorens[b], Javier Oliver[b], Josep Silva*[,b], Salvador Tamarit[b]

[a] *Institut für Informatik, Heinrich-Heine-Universität Düsseldorf, Universitätsstrasse 1, D-40225 Düsseldorf, Germany*
[b] *Universitat Politècnica de València, Camino de Vera S/N, E-46022 Valencia, Spain*

Abstract

Static analysis of concurrent languages is a complex task due to the non-deterministic execution of processes. If the concurrent language being studied allows process synchronization, then the analyses are even more complex (and thus expensive), e.g., due to the phenomenon of *deadlock*. In this work we introduce a static analysis technique based on program slicing for concurrent and explicitly synchronized languages in general, and CSP in particular. Concretely, given a particular point in a specification, our technique allows us to know what parts of the specification must necessarily be executed before this point, and what parts of the specification could be executed before it. Our technique is based on a new data structure that extends the *Synchronized Control Flow Graph* (SCFG). We show that this new data structure improves the SCFG by taking into account the context in which processes are called and, thus, it makes the slicing process more precise. The technique has been implemented and tested with real specifications, producing good results. After formally defining our technique, we describe our tool, its architecture, its main applications and the results obtained from several experiments conducted in order to measure the performance of the tool.

Key words: Concurrent Programming, CSP, Program Slicing

1. Introduction

Process algebras such as CSP [10], π-calculus [20] or LOTOS [2] and process modeling languages such as Promela [11, 22] allow us to specify complex systems with multiple interacting processes. The study and transformation of such systems often implies different analyses (e.g., deadlock analysis [15], reliability analysis [12], refinement checking [25], etc.).

☆This work has been partially supported by the Spanish *Ministerio de Economía y Competitividad (Secretaría de Estado de Investigación, Desarrollo e Innovación)* under grant TIN2008-06622-C03-02 and by the *Generalitat Valenciana* under grant PROMETEO/2011/052. Salvador Tamarit was partially supported by the Spanish MICINN under FPI grant BES-2009-015019.

*Corresponding author

Email addresses: leuschel@cs.uni-duesseldorf.de (Michael Leuschel), mllorens@dsic.upv.es (Marisa Llorens), fjoliver@dsic.upv.es (Javier Oliver), jsilva@dsic.upv.es (Josep Silva), stamarit@dsic.upv.es (Salvador Tamarit)

In this work we introduce a static analysis technique for process algebras with explicit synchronization mechanisms, based on a well-known program comprehension technique called *program slicing* [28]. Program slicing is a method for decomposing programs by analyzing their data and control flow. Roughly speaking, a *program slice* consists of those parts of a program that are (potentially) determining the values computed at some program point and/or variable, referred to as a *slicing criterion*. Program slices are usually computed from a *Program Dependence Graph* (PDG) [7] that makes explicit both the data and control dependences for each operation in a program. Program dependences can be traversed backwards or forwards (from the slicing criterion), that is known as *backward* or *forward* slicing, respectively. Additionally, slices can be *dynamic* or *static*, depending on whether a concrete program's input is provided or not. A survey on program slicing can be found, e.g., in [26].

Our technique allows us to extract the part of a specification related to a given point (referred to as the slicing criterion) in the specification. This technique can be very useful to debug, understand, maintain and reuse specifications; but also as a preprocessing stage of other analyses and/or transformations in order to reduce the complexity of the specification. In particular, given a point (e.g., an event) in a specification, our technique allows us to extract those parts of the specification that must be executed before the specified point (thus they are an implicit precondition); and those parts of the specification that could be executed before it. Therefore, the other parts of the specification cannot be executed before this point.

Example 1. *Consider the following specification*[1]*:*

```
MAIN = (STUDENT  ||  PARENT)   ||   COLLEGE
             {pass}      {pass,fail}

STUDENT = year1 → (pass → YEAR2 □ fail → STUDENT)

YEAR2 = year2 → (pass → YEAR3 □ [fail] → YEAR2)

YEAR3 = year3 → (pass → graduate → STOP □ fail → YEAR3)

PARENT = pass → present → PARENT

COLLEGE = fail → COLLEGE □ pass → C1

C1 = fail → COLLEGE □ pass → C2

C2 = fail → COLLEGE □ pass → prize → STOP
```

In this specification we have three processes (`STUDENT`*,* `PARENT` *and* `COLLEGE`*) executed in parallel and synchronized on common events. Process* `STUDENT` *represents the three-year academic courses of a student; process* `PARENT` *represents the parent of the student who gives her a present when she passes a course; and process* `COLLEGE` *represents the college who gives a prize to those students that finish without any fail.*

We are interested in determining what parts of the specification must be executed before the student fails in the second year, hence, we mark event `fail` *of process* `YEAR2`

[1] In the following, without lack of generality, we will use the *Communicating Sequential Processes* (CSP) [10] language as the running language for our examples. We refer those readers non familiar with CSP syntax to Section 2 where we provide a brief introduction to CSP.

(thus the slicing criterion is (YEAR2, fail)*, marked by a box in the above figure). Our slicing technique automatically extracts the slice consisting of the expressions in black. We can additionally be interested in knowing what parts could be executed before the same event. In this case, our technique adds to the slice the underscored parts because they could be executed (in some executions) before the marked event (observe that the result of this analysis is always a superset of the result obtained by the previous analysis). Therefore, this analysis could be used for program comprehension. Note, for instance, that in order to fail in the second year, the student has necessarily passed the first year. But, the parent may or may not have given a present to his daughter (even if she passed the first year) because this specification does not force the parent to give a present to his daughter until she has passed the second year. Moreover, note that the choice of process* C1 *belongs also to the slice. This is due to the fact that the slicing criterion must synchronize with the event* fail *of this process; therefore, the choice must be executed before the slicing criterion.*[2] *This is not so obvious from the specification, and the slice can help to understand the actual meaning of the specification.*

Computing the parts of the specification that could be executed before the slicing criterion can be useful, e.g., for debugging. If the slicing criterion is an event that executed incorrectly (i.e., it should not happen in the execution), then the slice produced contains all the parts of the specification that could produce the wrong behavior.

*A third application is program specialization. Note that the slices produced are not executable, but, in both cases, the slices could be made executable by replacing the removed parts by "*STOP*" or by "→ *STOP*" if the removed expression has a prefix. Hence, we have defined a further transformation that allows us to extract executable slices. The specialized specification contains all the necessary parts of the original specification whose execution leads to the slicing criterion (and then, the specialized specification finishes).*

We have implemented our technique producing the first program slicer for CSP specifications. In our implementation, the slicing process is completely automatic. Once the user has loaded a specification, she can select (with the mouse) the point she is interested in. Obviously, this simple action is enough to define a slicing criterion because the tool can automatically determine the process and the source position of interest. This implementation is a tool that has been integrated in the system ProB [16, 4], an animator and model checker for B and CSP. We will describe this tool in Section 5.

It should be clear that computing the minimum slice of an arbitrary CSP specification is an undecidable problem. Consider for instance the following CSP specification:

```
MAIN = P ⊓ Q

P = X ; Q

Q = a → STOP

X = Infinite Process
```

together with the slicing criterion (Q, a). Determining whether X does not belong to the slice implies determining whether X terminates, which is undecidable.

The main contributions of this work are the following:

[2] We could have chosen also to include the fail event of C1 into the slice. This is a matter of taste.

- We define two new static analyses for process algebras and propose algorithms for their implementation. Despite their clear usefulness we have not found similar static analyses in the literature.
- We define the *context-sensitive synchronized control flow graph* and show its advantages over its predecessors. This is a new data structure able to represent all computations of a specification taking into account the context of process calls; and it is particularly interesting for slicing languages with explicit synchronization.
- We have implemented our technique and integrated it in ProB [16, 4, 17]. Current releases of ProB are distributed with the slicer as an analysis tool. We present the implementation and the results obtained with several benchmarks.

The rest of the paper is organized as follows. In Section 2 we give an overview of the syntax and semantics of a process algebra (CSP) and introduce some notation that will be used along the article. In this section we also introduce an extension of the standard operational semantics of CSP. In Section 3 we show that previous data structures used in program slicing are inaccurate or inappropriate in our context, and we introduce the *Context-sensitive Synchronized Control Flow Graph* (CSCFG) as a solution and discuss its advantages over its predecessors. Our slicing technique is presented in Section 4 where we introduce two algorithms to slice CSP specifications from their CSCFGs. In Section 5 we present our implementation, we describe the architecture of our tool SOC, and we show the results of some experiments that reflect the efficiency and performance of the tool. Next, we discuss some related work in Section 6 and, finally, Section 7 concludes. All proofs of technical results can be found in Appendix A.

2. Communicating Sequential Processes

In order to keep the paper self-contained, in this section we recall the syntax and the semantics of the constructs used in our process algebra specifications. We use the CSP language [10], but the concepts and algorithms can also be applied to other process algebras. We also introduce here some notation that will be used along the paper.

Figure 1 summarizes the syntax constructions used in our CSP specifications. More precisely, a specification $\mathcal{S}$ is a finite collection of definitions. The left-hand side of each definition is the name of a different process, that is defined in the right-hand side (*rhs*) by means of an expression[3] that can be a call to another process or a combination of the following operators:

Prefixing. It specifies that event x (called the prefix) must happen before P.

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

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

[3] Therefore a process is defined by an expression, and thus, we often use indistinguishably these terms.

$\mathcal{S}$	$::=$	$D_1 \dots D_m$	(entire specification)	*Domains* $M, N, O \dots \in \mathcal{N}ames$ (names) $P, Q, R \dots \in \mathcal{P}$ (processes) $a, b, c \dots \in \Sigma$ (events) $u, v, w \dots \in \mathcal{V}$ (variables)
D	$::=$	$M = P$	(process definition)	
	$\mid$	$M(\overline{x_n}) = P$	(parameterized process)	where $\overline{x_n} = x_1, \dots, x_n$ and $x_i \in \Sigma \cup \mathcal{V}$
P	$::=$	M	(process call)	
	$\mid$	$M(\overline{x_n})$	(parameterized process call)	
	$\mid$	$x \to P$	(prefixing)	
	$\mid$	$c?u \to P$	(input)	
	$\mid$	$c!u \to P$	(output)	
	$\mid$	$P \sqcap Q$	(internal choice)	
	$\mid$	$P \square Q$	(external choice)	
	$\mid$	$P \nleftarrow bool \nrightarrow Q$	(conditional choice)	where $bool \in \{true, false\}$
	$\mid$	$P \mid\mid\mid Q$	(interleaving)	
	$\mid$	$P \underset{X}{\parallel} Q$	(synchronized parallelism)	where $X \subseteq \Sigma \cup \mathcal{V}$
	$\mid$	$P \; ; \; Q$	(sequential composition)	
	$\mid$	$P \backslash X$	(hiding)	
	$\mid$	$P[\![f]\!]$	(renaming)	where $f : (\Sigma \cup \mathcal{V}) \to (\Sigma \cup \mathcal{V})$
	$\mid$	$SKIP$	(skip)	
	$\mid$	$STOP$	(stop)	

Figure 1: Syntax of CSP specifications

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

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

Conditional choice. It is a choice that depends on a condition, i.e., it is equivalent to `if` *bool* `then` P `else` Q.

Interleaving. Both expressions are executed in parallel and independently.

Synchronized parallelism. Both expressions are executed in parallel with a set of synchronized events. In absence of synchronization both expressions can execute in any order. Whenever a synchronized event $x_i, 1 \leqslant i \leqslant n$, happens in one of the expressions it must also happen in the other at the same time. Whenever the set of synchronized events is not specified, it is assumed that the expressions are synchronized in all common events.

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

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

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

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

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

Figure 2 shows the standard operational semantics of CSP as defined by A. W. Roscoe [24]. This semantics is a logical inference system where a state is formed by a single expression called the *control*. The system starts with an initial state, and the rules of the semantics are used to infer how this state evolves. When no rules can be applied to the current state, the computation finishes. The rules of the semantics change the states of the computation due to the occurrence of events. The set of possible events is $\Sigma \cup \{\tau, \checkmark\}$. Events in $\Sigma = \{a, b, c \ldots\}$ are visible from the external environment, and can only happen with its cooperation (e.g., actions of the user). The special event τ cannot be observed from outside the system and it happens automatically as defined by the semantics. $\checkmark$ is a special event representing the successful termination of a process. The special symbol $\top$ is used to denote any process that already terminated.

The intuitive meaning of each rule is the following:

((Parameterized) Process Call) The call is unfolded and the right-hand side of process M is added to the control.

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

(SKIP) After `SKIP`, the only possible event is $\checkmark$, that denotes the end of the (sub)computation with the special symbol $\top$. There is no rule for $\top$ (nor for `STOP`), hence, this (sub)computation has finished.

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

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

(Conditional Choice 1 and 2) The condition *bool* is evaluated. If it is $true$, process P is put in the control, if it is $false$, process Q is.

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

(Process Call)

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

(Parameterized Process Call)

$$\frac{}{M(\overline{y_n}) \xrightarrow{\tau} rhs'(M)}$$

where $M(\overline{x_n}) = rhs(M) \in \mathcal{S}$ with $\overline{x_n}, \overline{y_n} \in \Sigma \cup \mathcal{V}$ and $rhs'(M) = rhs(M)$ with x_i replaced by $y_i, 1 \leqslant i \leqslant n$

(Prefixing)

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

(SKIP)

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

(Internal Choice 1)

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

(Internal Choice 2)

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

(External Choice 1)

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

(External Choice 2)

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

(External Choice 3)

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

(External Choice 4)

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

(Conditional Choice 1)

$$\frac{}{(P \not\leqslant true \not\geqslant Q) \xrightarrow{\tau} P}$$

(Conditional Choice 2)

$$\frac{}{(P \not\leqslant false \not\geqslant Q) \xrightarrow{\tau} Q}$$

(Synchronized Parallelism 1)

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

(Synchronized Parallelism 2)

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

(Synchronized Parallelism 3)

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

(Synchronized Parallelism 4)

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

Figure 2: CSP's operational semantics

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

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

(Sequential Composition 1)	(Sequential Composition 2)	
$\dfrac{P \xrightarrow{a\ or\ \tau} P'}{(P;Q) \xrightarrow{a\ or\ \tau} (P';Q)}$	$\dfrac{P \xrightarrow{\checkmark} \top}{(P;Q) \xrightarrow{\tau} Q}$	
(Hiding 1)	(Hiding 2)	(Hiding 3)
$\dfrac{P \xrightarrow{a} P'}{(P \backslash B) \xrightarrow{\tau} (P' \backslash B)}\ a \in B$	$\dfrac{P \xrightarrow{a\ or\ \tau} P'}{(P \backslash B) \xrightarrow{a\ or\ \tau} (P' \backslash B)}\ a \notin B$	$\dfrac{P \xrightarrow{\checkmark} \top}{(P \backslash B) \xrightarrow{\checkmark} \top}$
(Renaming 1)	(Renaming 2)	(Renaming 3)
$\dfrac{P \xrightarrow{a} P'}{(P[\![R]\!]) \xrightarrow{b} (P'[\![R]\!])}\ a\ R\ b$	$\dfrac{P \xrightarrow{a\ or\ \tau} P'}{(P[\![R]\!]) \xrightarrow{a\ or\ \tau} (P'[\![R]\!])}\ a\ R\ a$	$\dfrac{P \xrightarrow{\checkmark} \top}{(P[\![R]\!]) \xrightarrow{\checkmark} \top}$

Figure 2: CSP's operational semantics (cont.)

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

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

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

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

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

(Renaming 2 and 3) Renaming has no effect on either events renamed to themselves ($a\ R\ a$), and τ or $\checkmark$ events. The rules for renaming are similar to those for hiding.

We illustrate the semantics with the following example.

Example 2. *Consider the following CSP specification:*

$$\texttt{MAIN = (a} \rightarrow \texttt{STOP)} \underset{\{\texttt{a}\}}{\parallel} \texttt{(P} \;\square\; \texttt{(a} \rightarrow \texttt{STOP))}$$
$$\texttt{P = b} \rightarrow \texttt{SKIP}$$

If we use rhs(`MAIN`*) as the initial state to execute the semantics, we get the computation (i.e., sequence of valid state transitions) shown in Figure 3 where the final state is* $((\texttt{a} \rightarrow \texttt{STOP}) \underset{\{\texttt{a}\}}{\parallel} \top)$. *This computation corresponds to the execution of the left branch of the choice (i.e.,* `P`*) and thus only event* `b` *occurs. Each rewriting step is labeled with the applied rule.*

We need to define the notion of *specification position* that, roughly speaking, is a label that identifies a part of the specification. Formally,

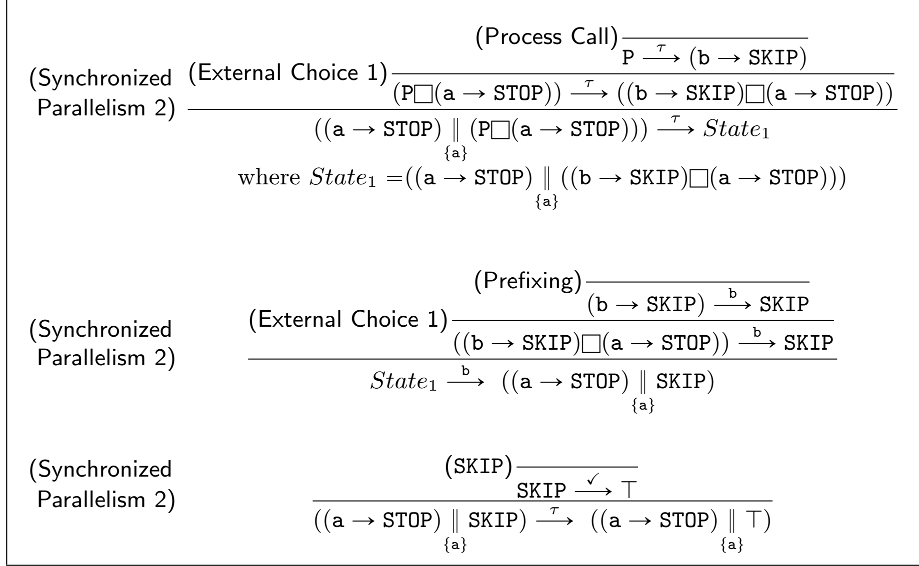


Figure 3: A computation with the operational semantics in Figure 2

Definition 1. (Position, Specification Position) Given a CSP specification $\mathcal{S}$ and a process definition $M = P$ in $\mathcal{S}$, the *positions* in each P are represented by a sequence of natural numbers, where Λ denotes the empty sequence (i.e., the root position). They are used to address the literals of an expression viewed as a tree, and are inductively defined as follows:

$P|_\Lambda = P$
$(P\ op)|_{1.w} = P|_w \quad \forall\ op \in \{\backslash, [\![]\!]\}$
$(P\ op\ Q)|_{1.w} = P|_w \quad \forall\ op \in \{\rightarrow, \sqcap, \square, \nleftrightarrow, |||, ||, ;\}$
$(P\ op\ Q)|_{2.w} = Q|_w \quad \forall\ op \in \{\rightarrow, \sqcap, \square, \nleftrightarrow, |||, ||, ;\}$

$P|_w$ is undefined otherwise.

A *specification position* is a pair (M, w) with $M = P \in \mathcal{S}$ and w a sequence of naturals, such that $P|_w$ is defined. We use the special specification position $(M, 0)$ for the left-hand side of the process definition $M = P$. We let $\mathcal{P}os(\mathcal{S})$ denote the set of all specification positions for processes in $\mathcal{S}$.

In the following we will refer to the literal associated to a specification position α with $lit(\alpha)$. For instance, in the specification of Example 3 where expressions are labeled with their associated specification positions, $lit((\texttt{MAIN}, 1)) = ||$ and $lit((\texttt{MAIN}, 1.1)) = \texttt{BUS}$. As we will work with graphs whose nodes are labeled with specification positions, we often use $\mathcal{P}os(N)$ to denote the set of all specification positions associated with the set of nodes N.

Example 3. *In the following specification*[4] $\mathcal{S}$ *each expression has been labeled (in grey color) with its associated specification position so that all labels are unique.*

[4]This is a simplification of a benchmark by Simon Gay to simulate a bus line.

$$\texttt{MAIN}_{(\texttt{MAIN},0)} = (\texttt{BUS}_{(\texttt{MAIN},1.1)} ||_{(\texttt{MAIN},1)} \texttt{P1}_{(\texttt{MAIN},1.2)}) ;_{(\texttt{MAIN},\Lambda)} (\texttt{BUS}_{(\texttt{MAIN},2.1)} ||_{(\texttt{MAIN},2)} \texttt{P2}_{(\texttt{MAIN},2.2)})$$

$$\texttt{BUS}_{(\texttt{BUS},0)} = \texttt{board}_{(\texttt{BUS},1)} \rightarrow_{(\texttt{BUS},\Lambda)} \texttt{alight}_{(\texttt{BUS},2.1)} \rightarrow_{(\texttt{BUS},2)} \texttt{SKIP}_{(\texttt{BUS},2.2)}$$

$$\texttt{P1}_{(\texttt{P1},0)} = \texttt{wait}_{(\texttt{P1},1)} \rightarrow_{(\texttt{P1},\Lambda)} \texttt{board}_{(\texttt{P1},2.1)} \rightarrow_{(\texttt{P1},2)} \texttt{alight}_{(\texttt{P1},2.2.1)} \rightarrow_{(\texttt{P1},2.2)} \texttt{SKIP}_{(\texttt{P1},2.2.2)}$$

$$\texttt{P2}_{(\texttt{P2},0)} = \texttt{wait}_{(\texttt{P2},1)} \rightarrow_{(\texttt{P2},\Lambda)} \texttt{board}_{(\texttt{P2},2.1)} \rightarrow_{(\texttt{P2},2)} \texttt{pay}_{(\texttt{P2},2.2.1)} \rightarrow_{(\texttt{P2},2.2)} \texttt{alight}_{(\texttt{P2},2.2.2.1)} \rightarrow_{(\texttt{P2},2.2.2)} \texttt{SKIP}_{(\texttt{P2},2.2.2.2)}$$

The notion of specification position allows us to determine what parts of the specification are executed in a particular execution. For this purpose, we have extended the semantics of Figure 2 in such a way that given a specification $\mathcal{S}$ and an execution of $\mathcal{S}$ with the extended semantics, the semantics produces as a side-effect the collection of specification positions that have been executed in this particular execution.

The extended semantics is presented in Figure 4 where we assume that every expression in the program has been labeled with its specification position (denoted by a subscript, e.g., P_α). A *state* of the semantics is a tuple (P, ω) where P is the *control*, i.e., the expression to be evaluated and ω represents the set of specification positions already evaluated. When the computation has finished or interrupted, ω contains the portion of the source code that has been executed.

An explanation for each rule of the semantics follows:

((Parameterized) Process Call) The called process is unfolded and its specification position α is added to the current set of specification positions ω. The new expression in the control is $rhs(M)$.

(Prefixing) Set ω is increased with the specification positions of the prefix and the prefixing operator.

(SKIP and STOP) The specification position α of `SKIP` (respectively `STOP`) is added to the current set of specification positions.

(Internal Choice 1 and 2) (Conditional Choice 1 and 2) The choice operator is added to ω.

(External Choice 1, 2, 3 and 4) External choices can develop both branches while τ events happen (rules 1 and 2), until an event in $\Sigma \cup \{\checkmark\}$ occurs (rules 3 and 4). This means that the semantics can develop both branches of the trace alternatively before selecting one branch. Of course, we want the extended semantics to collect all specification positions that have been executed and thus, when rules 1 and 2 are fired several times to evolve the branches of the choice, the corresponding specification positions are added to the common set ω.

(Synchronized Parallelism 1 and 2) Because nodes from both parallel processes can be executed interweaved, the parallelism operator is added to ω together with the specification positions (ω') executed of the corresponding branch.

(Synchronized Parallelism 3) When a synchronization occurs, the parallelism operator together with the specification positions executed in both branches are added to ω.

(Process Call)

$$\overline{(M_\alpha, \omega) \xrightarrow{\tau} (rhs(M), \omega \cup \{\alpha\})}$$

(Parameterized Process Call)

$$\overline{(M_\alpha(\overline{y_n}), \omega) \xrightarrow{\tau} (rhs'(M), \omega \cup \{\alpha\})}$$

where $M(\overline{x_n}) = rhs(M) \in \mathcal{S}$
with $\overline{x_n}, \overline{y_n} \in \Sigma \cup \mathcal{V}$
and $rhs'(M) = rhs(M)$
with x_i replaced by $y_i, 1 \leqslant i \leqslant n$

(Prefixing)

$$\overline{(a_\alpha {\rightarrow}_\beta P, \omega) \xrightarrow{a} (P, \omega \cup \{\alpha, \beta\})}$$

(SKIP)

$$\overline{(\texttt{SKIP}_\alpha, \omega) \xrightarrow{\checkmark} (\top, \omega \cup \{\alpha\})}$$

(STOP)

$$\overline{(\texttt{STOP}_\alpha, \omega) \xrightarrow{\tau} (\bot, \omega \cup \{\alpha\})}$$

(Internal Choice 1)

$$\overline{(P \sqcap_\alpha Q, \omega) \xrightarrow{\tau} (P, \omega \cup \{\alpha\})}$$

(Internal Choice 2)

$$\overline{(P \sqcap_\alpha Q, \omega) \xrightarrow{\tau} (Q, \omega \cup \{\alpha\})}$$

(External Choice 1)

$$\frac{(P, \omega) \xrightarrow{\tau} (P', \omega')}{(P \;\Box_\alpha\; Q, \omega) \xrightarrow{\tau} (P' \;\Box_\alpha\; Q, \omega' \cup \{\alpha\})}$$

(External Choice 2)

$$\frac{(Q, \omega) \xrightarrow{\tau} (Q', \omega')}{(P \;\Box_\alpha\; Q, \omega) \xrightarrow{\tau} (P \;\Box_\alpha\; Q', \omega' \cup \{\alpha\})}$$

(External Choice 3)

$$\frac{(P, \omega) \xrightarrow{a \text{ or } \checkmark} (P', \omega')}{(P \;\Box_\alpha\; Q, \omega) \xrightarrow{a \text{ or } \checkmark} (P', \omega' \cup \{\alpha\})}$$

(External Choice 4)

$$\frac{(Q, \omega) \xrightarrow{a \text{ or } \checkmark} (Q', \omega')}{(P \;\Box_\alpha\; Q, \omega) \xrightarrow{a \text{ or } \checkmark} (Q', \omega' \cup \{\alpha\})}$$

(Conditional Choice 1)

$$\overline{(P \not\leqslant true \not\geqslant_\alpha Q, \omega) \xrightarrow{\tau} (P, \omega \cup \{\alpha\})}$$

(Conditional Choice 2)

$$\overline{(P \not\leqslant false \not\geqslant_\alpha Q, \omega) \xrightarrow{\tau} (Q, \omega \cup \{\alpha\})}$$

(Synchronized Parallelism 1)

$$\frac{(P, \omega) \xrightarrow{a \text{ or } \{\tau \text{ or } \checkmark\}} (P', \omega')}{(P \underset{X_\alpha}{\|} Q, \omega) \xrightarrow{a \text{ or } \tau} (P' \underset{X}{\|} Q, \omega' \cup \{\alpha\})} \; a \notin X$$

(Synchronized Parallelism 2)

$$\frac{(Q, \omega) \xrightarrow{a \text{ or } \{\tau \text{ or } \checkmark\}} (Q', \omega')}{(P \underset{X_\alpha}{\|} Q, \omega) \xrightarrow{a \text{ or } \tau} (P \underset{X}{\|} Q', \omega' \cup \{\alpha\})} \; a \notin X$$

(Synchronized Parallelism 3)

$$\frac{(P, \omega) \xrightarrow{a} (P', \omega') \quad (Q, \omega) \xrightarrow{a} (Q', \omega'')}{(P \underset{X_\alpha}{\|} Q, \omega) \xrightarrow{a} (P' \underset{X}{\|} Q', \omega' \cup \omega'' \cup \{\alpha\})} \; a \in X$$

(Synchronized Parallelism 4)

$$\overline{(\top \underset{X}{\|} \top, \omega) \xrightarrow{\checkmark} (\top, \omega)}$$

Figure 4: An instrumented operational semantics for CSP with specification positions

(Sequential Composition 1)

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

(Sequential Composition 2)

$$\frac{(P,\omega) \xrightarrow{\checkmark} (\top,\omega')}{(P;_\alpha Q,\omega) \xrightarrow{\tau} (Q,\omega' \cup \{\alpha\})}$$

(Hiding 1)

$$\frac{(P,\omega) \xrightarrow{a} (P',\omega')}{(P\backslash_\alpha B,\omega) \xrightarrow{\tau} (P'\backslash_\alpha B,\omega' \cup \{\alpha\})}\ a \in B$$

(Hiding 2)

$$\frac{(P,\omega) \xrightarrow{a\ or\ \tau} (P',\omega')}{(P\backslash_\alpha B,\omega) \xrightarrow{a\ or\ \tau} (P'\backslash_\alpha B,\omega' \cup \{\alpha\})}\ a \notin B$$

(Hiding 3)

$$\frac{(P,\omega) \xrightarrow{\checkmark} (\top,\omega')}{(P\backslash_\alpha B,\omega) \xrightarrow{\checkmark} (\top,\omega' \cup \{\alpha\})}$$

(Renaming 1)

$$\frac{(P,\omega) \xrightarrow{a} (P',\omega')}{(P[\![_\alpha R]\!],\omega) \xrightarrow{b} (P'[\![_\alpha R]\!],\omega' \cup \{\alpha\})}\ a\ R\ b$$

(Renaming 2)

$$\frac{(P,\omega) \xrightarrow{a\ or\ \tau} (P',\omega')}{(P[\![_\alpha R]\!],\omega) \xrightarrow{a\ or\ \tau} (P'[\![_\alpha R]\!],\omega' \cup \{\alpha\})}\ a\ R\ a$$

(Renaming 3)

$$\frac{(P,\omega) \xrightarrow{\checkmark} (\top,\omega')}{(P[\![_\alpha R]\!],\omega) \xrightarrow{\checkmark} (\top,\omega' \cup \{\alpha\})}$$

Figure 4: An instrumented operational semantics for CSP with specification positions (cont.)

(Synchronized Parallelism 4) It has no influence over the set ω because the processes already terminated, and thus, the parallelism operator is already included in the set by the other rules.

(Sequential Composition 1 and 2) Sequential Composition 1 is used to add to ω the specification positions executed in process P until it is finished. When P finishes Sequential Composition 2 is used and the specification position of ; is added to ω.

(Hiding 1, 2 and 3) ω is increased with the specification position of the Hiding operator and the specification positions of the developed process P.

(Renaming 1, 2 and 3) It is completely analogous to the previous case.

Example 4. *Consider again the specification of Example 2 but now expressions are labeled with their associated specification positions (in grey color) so that labels are unique.*

$$\texttt{MAIN}_{(\texttt{MAIN},0)} = (\texttt{a}_{(\texttt{MAIN},1.1)} \rightarrow_{(\texttt{MAIN},1)} \texttt{STOP}_{(\texttt{MAIN},1.2)}) \underset{\{\texttt{a}\}}{\|}_{(\texttt{MAIN},\Lambda)}$$
$$(\texttt{P}_{(\texttt{MAIN},2.1)} \square_{(\texttt{MAIN},2)} (\texttt{a}_{(\texttt{MAIN},2.2.1)} \rightarrow_{(\texttt{MAIN},2.2)} \texttt{STOP}_{(\texttt{MAIN},2.2.2)}))$$

$$\texttt{P}_{(\texttt{P},0)} = \texttt{b}_{(\texttt{P},1)} \rightarrow_{(\texttt{P},\Lambda)} \texttt{SKIP}_{(\texttt{P},2)}$$

The execution of the instrumented semantics in Figure 4 with the initial state $(rhs(\texttt{MAIN}), \emptyset)$ *produces the computation of Figure 5. Here, for clarity, each computation step is labeled with the applied rule (*EC 4 *means* External Choice 4*); in each state, the second component denotes the set of specification positions already evaluated. Note that the first rule applied is (*Synchronized Parallelism 3*) to the initial expression* $rhs(\texttt{MAIN})$*. This computation corresponds to the execution of the right branch of the choice (i.e.,* $\mathsf{a} \rightarrow \texttt{STOP}$*). The final state is* $(\bot \underset{\{\mathtt{a}\}}{\|} \bot,\ \omega_4)$ *where* $\omega_4 = \{(\texttt{MAIN}, \Lambda), (\texttt{MAIN}, 1.1), (\texttt{MAIN}, 1), (\texttt{MAIN}, 1.2), (\texttt{MAIN}, 2), (\texttt{MAIN}, 2.2.1), (\texttt{MAIN}, 2.2), (\texttt{MAIN}, 2.2.2)\}$*.*

$$(\text{Synchronized Parallelism 3})\frac{L \quad R}{State\ 1 \xrightarrow{a} State\ 2} \text{ where}$$

$$State\ 1 = ((\mathsf{a} \rightarrow \texttt{STOP}) \underset{\{\mathtt{a}\}}{\|}_{(\texttt{MAIN},\Lambda)} (\mathsf{P}\Box(\mathsf{a} \rightarrow \texttt{STOP})), \emptyset)$$

$$L = (\text{Prefixing})\frac{}{(\mathsf{a} \rightarrow \texttt{STOP}, \emptyset) \xrightarrow{\mathsf{a}} (\texttt{STOP}, \omega_0)}$$
$$\text{where } \omega_0 = \{(\texttt{MAIN}, 1.1), (\texttt{MAIN}, 1)\}$$

$$R = (\text{EC 4})\frac{(\text{Prefixing})\frac{}{(\mathsf{a} \rightarrow \texttt{STOP}, \emptyset) \xrightarrow{\mathsf{a}} (\texttt{STOP}, \{(\texttt{MAIN}, 2.2.1), (\texttt{MAIN}, 2.2)\})}}{((\mathsf{P}\Box_{(\texttt{MAIN},2)}(\mathsf{a} \rightarrow \texttt{STOP})), \emptyset) \xrightarrow{\mathsf{a}} (\texttt{STOP}, \omega_1)}$$
$$\text{where } \omega_1 = \{(\texttt{MAIN}, 2.2.1), (\texttt{MAIN}, 2.2)\} \cup \{(\texttt{MAIN}, 2)\}$$

$$\text{and } State\ 2 = ((\texttt{STOP} \underset{\{\mathtt{a}\}}{\|} \texttt{STOP}), \omega_2) \text{ where } \omega_2 = \omega_0 \cup \omega_1 \cup \{(\texttt{MAIN}, \Lambda)\}$$

$$(\text{Synchronized Parallelism 1})\frac{(\texttt{STOP})\frac{}{(\texttt{STOP}, \omega_2) \xrightarrow{\tau} (\bot, \omega_2 \cup \{(\texttt{MAIN}, 1.2)\})}}{State\ 2 \xrightarrow{\tau} State\ 3}$$

$$\text{where } State\ 3 = (\bot \underset{\{\mathtt{a}\}}{\|} \texttt{STOP}, \omega_3) \text{ and } \omega_3 = \omega'' \cup \{(\texttt{MAIN}, 1.2)\} \cup \{(\texttt{MAIN}, \Lambda)\}$$

$$(\text{Synchronized Parallelism 2})\frac{(\texttt{STOP})\frac{}{(\texttt{STOP}, \omega_3) \xrightarrow{\tau} (\bot, \omega_3 \cup \{(\texttt{MAIN}, 2.2.2)\})}}{State\ 3 \xrightarrow{\tau} State\ 4}$$

$$\text{where } State\ 4 = (\bot \underset{\{\mathtt{a}\}}{\|} \bot, \omega_4) \text{ and } \omega_4 = \omega_3 \cup \{(\texttt{MAIN}, 2.2.2)\} \cup \{(\texttt{MAIN}, \Lambda)\}$$

Figure 5: An example of computation with the semantics in Figure 4

Definition 2. (Rewriting Step, Derivation) Given a state of the semantics s, a *rewriting step* for s[5] is the application of a rule of the semantics: $\dfrac{\Theta}{s \xrightarrow{a\ or\ \tau\ or\ \checkmark} s'}$ where Θ is a (possibly empty) set of rewriting steps. We say that the rewriting step is *simple* iff Θ is

[5]Note that because s is a state, this definition is valid for both semantics presented so far.

empty. For the sake of concreteness, we often represent the rewriting step for s as $(s \longrightarrow s')$. Given a state of the semantics s_0, we say that the sequence $s_0 \longrightarrow \ldots \longrightarrow s_{n+1}$, $n \geqslant 0$, is a *derivation* of s_0 iff $\forall\ i, 0 \leqslant i \leqslant n, s_i \longrightarrow s_{i+1}$ 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 the derivation has *successfully finished* iff the control of s_{n+1} is $\top$.

We use $s_1 \longrightarrow^* s_n$ to denote a feasible (sub)derivation $s_0 \longrightarrow \ldots \longrightarrow s_n$ that leads from s_1 to s_n; and we define $\mathcal{P}os(s_1 \longrightarrow^* s_n) = \{\mathcal{P}os(c_i) \mid 1 \leqslant i \leqslant n\}$ where c_i is the control of state s_i. In the following, we will assume that computations start from a distinguished process `MAIN`.

We also define the following notation for a given CSP specification $\mathcal{S}$: $\mathcal{C}alls(\mathcal{S})$ is the set of specification positions for the process calls appearing in $\mathcal{S}$. $\mathcal{P}roc(\mathcal{S})$ is the set of specification positions in left-hand sides of the processes in $\mathcal{S}$ (i.e., $\mathcal{P}roc(\mathcal{S}) = \{\alpha \in \mathcal{P}os(\mathcal{S}) \mid \alpha = (M, 0)\}$).

In addition, given a set of specification positions A, we define *choices*(A) as the subset of specification positions of operators that are either an internal choice, an external choice or a conditional choice. For instance, in the specification $\mathcal{S}$ of Example 3 we have $\mathcal{C}alls(\mathcal{S}) = \{(\mathtt{MAIN}, 1.1), (\mathtt{MAIN}, 1.2), (\mathtt{MAIN}, 2.1), (\mathtt{MAIN}, 2.2)\}$ and $\mathcal{P}roc(\mathcal{S}) = \{(\mathtt{MAIN}, 0), (\mathtt{BUS}, 0), (\mathtt{P1}, 0), (\mathtt{P2}, 0)\}$.

3. Context-sensitive Synchronized Control Flow Graph

As usual in static analysis, we need a data structure capable of finitely representing the (often infinite) computations of our specifications. Unfortunately, we cannot use the standard *Control Flow Graph* (CFG) [26], nor the *Interprocedural Control Flow Graph* (ICFG) [9] because they cannot represent multiple threads and, thus, they can only be used with sequential programs. In fact, for CSP specifications, being able to represent multiple threads is a necessary but not a sufficient condition. For instance, the *threaded Control Flow Graph* (tCFG) [13, 14] can represent multiple threads through the use of the so called "*start thread*" and "*end thread*" nodes; but it does not handle synchronization between threads. Callahan and Sublok introduced in [5] the *Synchronized Control Flow Graph* (SCFG), a data structure proposed in the context of imperative programs where an event variable is always in one of two states: clear or posted. The initial value of an event variable is always clear. The value of an event variable can be set to posted with the *POST* statement; and a *WAIT* statement suspends execution of the thread that executes it until the specified event variableś value is set to posted. The SCFG explicitly represents synchronization between threads with a special edge for synchronization flows. In words by Callahan and Sublok [5]:

> "*A synchronized control flow graph is a control flow graph augmented with a set E_s of synchronization edges. $(b_1, b_2) \in E_s$ if the last statement in block b_1 is POST(ev) and the first statement in block b_2 is WAIT(ev) where ev is an event variable.*"

In order to adapt the SCFG to CSP, we extend it with the "*start thread*" and "*end thread*" notation from tCFGs. Therefore, in the following we will work with graphs where nodes N are labeled with positions and "*start*", "*end*" labels (we denote the label of node n with $l(n)$). We also use this notation, "*end* $\backslash$" and "*end* $[\![\,]\!]$", to denote the end of a

hiding respectively a renaming operator. In particular, $\forall n \in N, l(n) \in \mathcal{P}os(\mathcal{S}) \cup \mathcal{S}tart(\mathcal{S})$ where:

$$\begin{aligned}\mathcal{S}tart(\mathcal{S}) = &\{\text{“}start\ \alpha\text{”}, \text{“}end\ \alpha\text{”} \mid \alpha \in \mathcal{P}roc(\mathcal{S})\} \\ &\cup \{\text{“}end\ \alpha\text{”} \mid \alpha \in \mathcal{P}os(\mathcal{S}) \wedge \ lit(\alpha) \in \{\backslash, \llbracket\rrbracket\}\}\end{aligned}$$

For the definition of SCFG, we need to provide a notion of *control flow* between the nodes of a labeled graph.

Definition 3. (Control flow) Given a CSP specification $\mathcal{S}$ and a set of labeled nodes N such that $\forall n \in N, l(n) \in \mathcal{P}os(\mathcal{S}) \cup \mathcal{S}tart(\mathcal{S})$, the *control flow* is a binary relation between the nodes in N. Given two nodes n, $n' \in N$, we say that the *control* of n *can pass* to n' iff:

1. $lit(l(n)) \in \{\sqcap, \Box, \nless\ngtr, |||, ||\} \wedge l(n) = (M, w) \wedge l(n') \in \{first((M, w.1)), first((M, w.2))\}$
2. $lit(l(n')) = \rightarrow \wedge \ l(n') = (M, w) \ \wedge \ l(n) = (M, w.1)$
3. $lit(l(n')) = \ ; \wedge \ l(n') = (M, w) \ \wedge \ l(n) \in last((M, w.1))$
4. $lit(l(n)) \in \{\rightarrow, ;\} \ \wedge \ l(n) = (M, w) \ \wedge \ l(n') = first((M, w.2))$
5. $lit(l(n)) \in \{\backslash, \llbracket\rrbracket\} \ \wedge \ l(n) = (M, w) \ \wedge \ l(n') = first((M, w.1))$
6. $l(n') = \text{“}end\ (M, w)\text{”} \ \wedge \ lit((M, w)) \in \{\backslash, \llbracket\rrbracket\} \ \wedge \ l(n) \in last((M, w.1))$

where $first((M, w))$ is defined as follows:

$$first((M, w)) = \begin{cases} (M, w.1) & \text{if } lit((M, w)) = \ \rightarrow \\ first((M, w.1)) & \text{if } lit((M, w)) = \ ; \\ (M, w) & \text{otherwise} \end{cases}$$

and where $last((M, w))$ is the set of possible termination points of (M, w):

$$last((M, w)) = \begin{cases} \{(M, w)\} & \text{if } lit((M, w)) = SKIP \\ \varnothing & \text{if } lit((M, w)) = STOP \vee (lit((M, w)) \in \{|||, ||\} \wedge \\ & \qquad (last((M, w.1)) = \varnothing \vee last((M, w.2)) = \varnothing)) \\ last((M, w.1)) & \text{if } lit((M, w)) \in \{\sqcap, \Box, \nless\ngtr\} \vee (lit((M, w)) \in \{|||, ||\} \wedge \\ \cup\ last((M, w.2)) & \qquad last((M, w.1)) \neq \varnothing \wedge last((M, w.2)) \neq \varnothing) \\ last((M, w.2)) & \text{if } lit((M, w)) \in \{\rightarrow, ;\} \\ \{\text{“}end\ (M, w)\text{”}\} & \text{if } lit((M, w)) \in \{\backslash, \llbracket\rrbracket\} \end{cases}$$

Rather than using a declarative definition of SCFG, we provide a constructive definition based on the control flow that allows us to compute the SCFG from a CSP specification.

Definition 4. (Synchronized Control Flow Graph) Given a CSP specification $\mathcal{S}$, we define its *Synchronized Control Flow Graph* as a graph $\mathcal{G} = (N, E_c, E_s)$ where nodes $N = \mathcal{P}os(\mathcal{S}) \cup \mathcal{S}tart(\mathcal{S})$. Edges are divided into two groups, *control-flow arcs* (E_c) and *synchronization edges* (E_s). E_s is a set of edges (denoted by $\leftrightarrow$) representing the possible synchronization of two (event) nodes.[6] E_c is a set of arcs (denoted with $\mapsto$) such that, given two nodes $n, n' \in N$, $n \mapsto n' \in E_c$ iff the control of n can pass to n' or one of the following is true:

[6]Computing the events that will synchronize in a specification is a field of research by itself. There are many approaches and algorithms to do this task. In our implementation, we use the technique from [23].

- $lit(l(n)) = M \wedge l(n') =$ "*start* $(M, 0)$" with $l(n) \in \mathcal{C}alls(\mathcal{S})$
- $l(n) =$ "*start* $(M, 0)$" $\wedge\ l(n') = \mathit{first}((M, \Lambda))$
- $l(n) \in \mathit{last}((M, \Lambda)) \wedge l(n') =$ "*end* $(M, 0)$"

where $\mathit{last}((M, w))$ with $(M, w) \in \mathcal{C}alls(\mathcal{S})$ is defined as $\mathit{last}((M, w)) = \{$"*end* $(P, 0)$"$\}$.

Observe that the size of the SCFG is $\mathcal{O}(n)$ being n the number of positions in the specification. This can be easily proved by showing that there is only one node in the SCFG for each position of the specification, and specification positions are finite and unique. To be fully precise, there is exactly one node for each specification position and two extra nodes for each process (the start process and end process nodes) and one extra node for the hiding and renaming operators (the end hiding and the end renaming). Hence, the size of a SCFG associated to a specification with p processes and n positions with r hiding and renaming operators is $2p + n + r$. The SCFG can be used for slicing CSP specifications as it is described in the following example.

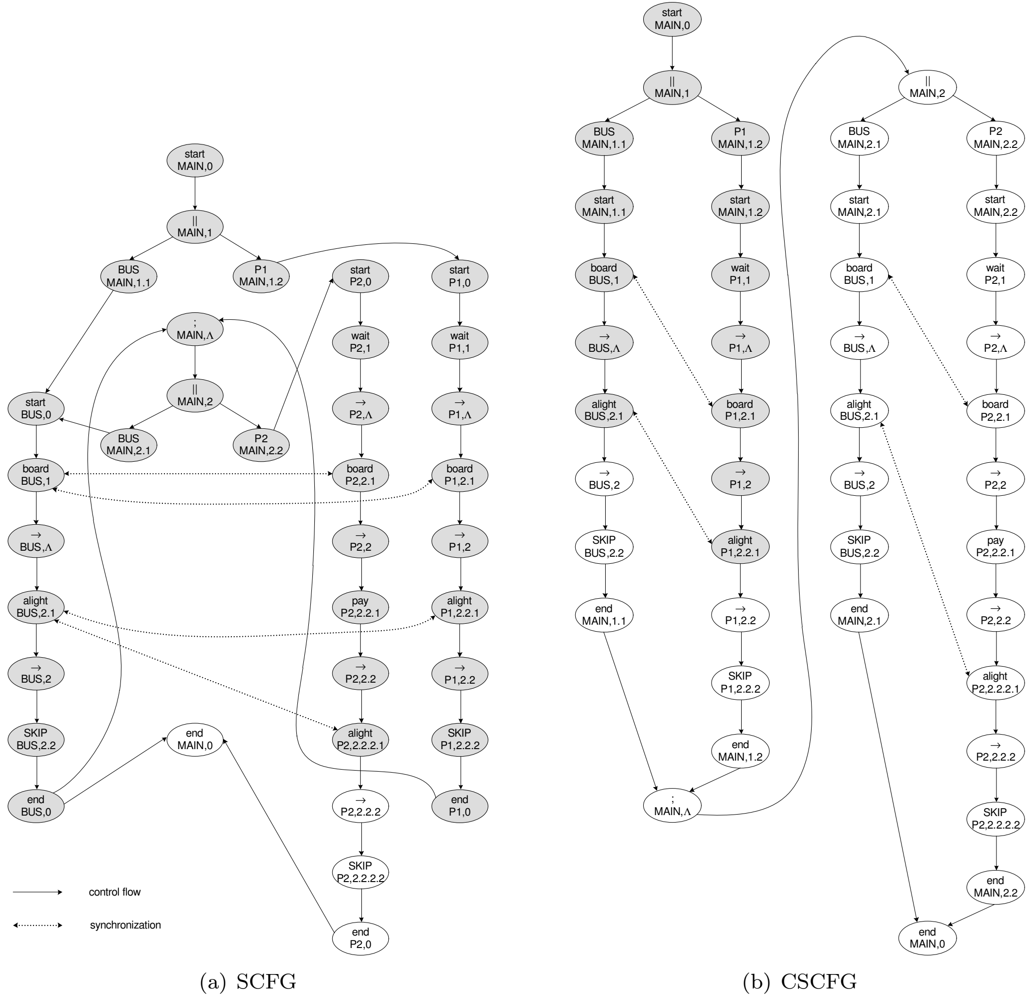
Example 5. *Consider the specification of Example 3 and its associated SCFG shown in Figure 6(a); for the sake of clarity we show the expression represented by each specification position. If we select the node labeled* `(P1,alight)` *and traverse the SCFG backwards in order to identify the nodes on which* `(P1,alight)` *depends, we get the grey nodes of the graph.*

The purpose of this example is twofold: on the one hand, it shows that the SCFG can be used for static slicing of CSP specifications. On the other hand, it shows that it is still too imprecise to be used in practice. The cause of this imprecision is that the SCFG is context-insensitive, because it connects all the calls to the same process with a unique set of nodes. This causes the SCFG to mix different executions of a process with possibly different synchronizations, and, thus it loses precision. For instance, in Example 3 process `BUS` is called twice in different contexts. It is first executed in parallel with `P1` producing the synchronization of their `board` and `alight` events. Then, it is executed in parallel with `P2` producing the synchronization of their `board` and `alight` events. This makes the process `P2` (except nodes $\rightarrow$, `SKIP` and `end P2`) be part of the slice. This is suboptimal because process `P2` is always executed after `P1`.

To the best of our knowledge, there do not exist other data structures that face the problem of representing concurrent and explicitly synchronized computations in a context-sensitive manner. In the rest of this section, we propose a new version of the SCFG, the context-sensitive synchronized control flow graph (CSCFG) which is context-sensitive because it takes into account the different contexts on which a process can be executed.

In contrast to the SCFG, the same specification position can appear multiple times inside a CSCFG. Hence, in the following we will use a refined notion of the $\mathcal{S}tart$ set so that in each "*start* α" and "*end* α" node used to represent a process, α is now any specification position representing a process call instead of a process definition (i.e., not necessarily $\alpha \in \mathcal{P}roc(\mathcal{S})$):

$$\begin{aligned}\mathcal{S}tart(\mathcal{S}) = {} & \{\text{"}\mathit{start}\ (\mathtt{MAIN}, 0)\text{"}, \text{"}\mathit{end}\ (\mathtt{MAIN}, 0)\text{"}\} \\ & \cup \{\text{"}\mathit{start}\ \alpha\text{"}, \text{"}\mathit{end}\ \alpha\text{"} \mid \alpha \in \mathcal{C}alls(\mathcal{S})\}\end{aligned}$$



(a) SCFG (b) CSCFG

Figure 6: SCFG and CSCFG of the program in Example 3

$$\cup\ \{\text{“}\mathit{end}\ \alpha\text{”} \mid \alpha \in \mathcal{P}os(\mathcal{S}) \land\ lit(\alpha) \in \{\backslash, [\![\,]\!]\}\}$$

Using the specification position of the process call allows us to distinguish between different process calls to the same process. Note that we also added to the set the initial and ending nodes of the graph (“*start* $(\mathtt{MAIN}, 0)$” and “*end* $(\mathtt{MAIN}, 0)$”).

Before we properly define the CSCFG, we provide a notion of path and context.

Definition 5. (Path) Given a labeled graph $\mathcal{G} = (N, E_c)$, a *path* between two nodes $n_1, n_k \in N$, represented by $n_1 \mapsto^* n_k$, is a sequence $l(n_1), \ldots, l(n_{k-1})$ such that for all $1 \leqslant i < k$ we have $n_i \mapsto n_{i+1} \in E_c$. The path is *loop-free* if for all $i \neq j$ we have $n_i \neq n_j$.

Definition 6. (Context) Given a labeled graph $\mathcal{G} = (N, E_c)$ and a node $n \in N$, the *context* of n, $\mathcal{C}on(n) = \{m \in N \mid l(m) = \text{“}\mathit{start}\ \alpha\text{”}, \alpha \in \mathcal{C}alls(\mathcal{S})$ and there exists a

loop-free path $\pi = m \mapsto^* n$ with "*end* α" $\notin \pi\}$.

Intuitively speaking, the context of a node represents the set of processes in which a particular node is being executed. If we focus on a node n with $l(n) \in \mathcal{C}alls(\mathcal{S})$ we can use the context to identify loops because we have a loop whenever "*start* $l(n)$" $\in \mathcal{C}on(n)$.

The main difference between the SCFG and the CSCFG is that the SCFG represents a process with a single collection of nodes (each specification position in the process is represented with a single node, see Figure 6(a)); in contrast, the CSCFG represents a process with multiple collections of nodes, each collection representing a different call to this process (i.e., a different context in which it is executed. For instance, see Figure 6(b) where process `BUS` is represented twice). Therefore, the notion of control flow used in the SCFG is insufficient for the CSCFG, and we need to extend it to also consider the context of process calls.

Definition 7. (Context-sensitive control flow) Given a CSP specification $\mathcal{S}$ and a labeled graph $\mathcal{G} = (N, E_c)$ such that $\forall n \in N, l(n) \in \mathcal{P}os(\mathcal{S}) \cup \mathcal{S}tart(\mathcal{S})$, the *context-sensitive control flow* is a binary relation between the nodes in N. Given two nodes n, $n' \in N$, we say that the *context-sensitive control* of n *can pass* to n', i.e., $n \mapsto n' \in E_c$, iff:

- the *control* of n *can pass* to n', or
- $lit(l(n')) = \ ; \wedge l(n') = (M, w) \wedge l(n) \in last(n'')$ with $n'' \in N \ \wedge \ l(n'') = (M, w.1)$
- $l(n') =$ "*end* (M, w)" $\wedge\ lit((M, w)) \in \{\backslash, [\![\,]\!]\} \wedge l(n) \in last(n'')$ with $n'' \in N \wedge l(n'') = (M, w.1)$

where $last(n)$ with $l(n) = (M, w)$ is the set of possible termination points of n:

$$last(n) = \begin{cases} \{(M,w)\} & \text{if } lit((M,w)) = SKIP \\ \varnothing & \text{if } lit((M,w)) = STOP \vee (lit((M,w)) \in \{|||, ||\} \wedge \\ & \quad (last(n1) = \varnothing \vee last(n2) = \varnothing)) \vee \\ & \quad (lit((M,w)) \in \mathcal{C}alls(\mathcal{S}) \wedge \text{"}start\ (M,w)\text{"} \in \mathcal{C}on(n)) \\ last(n1) & \text{if } lit((M,w)) \in \{\sqcap, \Box, \nleftarrow\!\!\nrightarrow\} \vee (lit((M,w)) \in \{|||, ||\} \wedge \\ \cup\ last(n2) & \qquad last(n1) \neq \varnothing \wedge last(n2) \neq \varnothing) \\ last(n2) & \text{if } lit((M,w)) \in \{\rightarrow, ;\} \\ \{\text{"}end\ (M,w)\text{"}\} & \text{if } lit((M,w)) \in \{\backslash, [\![\,]\!]\} \vee \\ & \quad (lit((M,w)) \in \mathcal{C}alls(\mathcal{S}) \wedge \text{"}start\ (M,w)\text{"} \notin \mathcal{C}on(n)) \end{cases}$$

where $l(n1) = (M, w.1)$ and $l(n2) = (M, w.2)$.

Definition 8. (Context-sensitive Synchronized Control Flow Graph) Given a CSP specification $\mathcal{S}$, we define its *Context-sensitive Synchronized Control Flow Graph* as a graph $\mathcal{G} = (N, E_c, E_l, E_s)$ where nodes N are labeled so that $\forall n \in N, l(n) \in \mathcal{P}os(\mathcal{S}) \cup \mathcal{S}tart(\mathcal{S})$; and $\mathcal{S}tart(\mathcal{S}) = \{$"*start* $(\texttt{MAIN}, 0)$", "*end* $(\texttt{MAIN}, 0)$"$\} \cup \{$"*start* α", "*end* α"$\mid \alpha \in \mathcal{C}alls(\mathcal{S})\} \cup \{$"*end* α"$\mid lit(\alpha) \in \{\backslash, [\![\,]\!]\}\}$. Edges are divided into three groups, *control-flow arcs* (E_c), *loop arcs* (E_l) and *synchronization edges* (E_s).

- E_s is a set of edges (denoted by $\leftrightarrow$) representing the possible synchronization of two (event) nodes.[6]

- E_c is a set of arcs (denoted by $\mapsto$) such that, given two nodes $n, n' \in N$, $n \mapsto n' \in E_c$ iff the *context-sensitive control* of n can pass to n' or $l(n) \in \mathcal{C}alls(\mathcal{S}) \wedge l(n') =$ "*start* $l(n)$". And given three nodes $n_1, n_2, n_5 \in N$:
  - $(n_1 \mapsto n_2) \in E_c$ iff $l(n_1) =$ "*start* α" $\wedge\ lit(\alpha) = M \wedge l(n_2) = \mathit{first}((M, \Lambda))$,
  - if $l(n_1) \in \mathcal{C}alls(\mathcal{S})$, $l(n_2) =$ "*start* $l(n_1)$" and $n_2 \in \mathcal{C}on(n_1)$ then $\forall n_4 \in N, (n_4 \mapsto n_5)$ with $l(n_4) \in \mathit{last}(n_3)$ with $n_2 \mapsto n_3 \wedge\ l(n_5) =$ "*end* α", and
  - $(n_1 \mapsto n_2) \in E_c$ iff $l(n_1) \in \mathit{last}((\mathtt{MAIN}, \Lambda)) \wedge l(n_2) =$ "*end* $(\mathtt{MAIN}, 0)$".
- E_l is a set of edges (denoted by $\rightsquigarrow$) used to represent loops, i.e., $(n_1 \rightsquigarrow n_2) \in E_l$ iff $l(n_1) \in \mathcal{C}alls(\mathcal{S})$, $l(n_2) =$ "*start* $l(n_1)$" and $n_2 \in \mathcal{C}on(n_1)$.

The CSCFG satisfies the following properties: (i) Two nodes can have the same label. (ii) Every node whose label belongs to {"*start* α"| $\alpha \in \mathcal{C}alls(\mathcal{S})$} has one and only one incoming arc in E_c. (iii) Every process call node has one and only one outgoing arc that belongs to either E_c or E_l.

The key difference between the SCFG and the CSCFG is that the latter unfolds every process call node except those that belong to a loop. This is very convenient for slicing because every process call that is executed in a different context is unfolded and represented with a different subgraph, thus, slicing does not mix computations. Moreover, it allows us to deal with recursion and, at the same time, it prevents infinite unfolding of process calls thanks to the use of loop arcs. Note that loop arcs are only used when the context is repeated (this is ensured by item 3 of the definition). Note also that loops are unfolded only once because the second time they are going to be unfolded the context of the process call node is repeated, and thus a loop arc is used to prevent the unfolding. Properties 2 and 3 ensure finiteness because process calls only have one outgoing arc, and thus, they cannot have a control arc if there is already a loop arc.

The following lemma ensures that the CSCFG is complete: all possible derivations of a CSP specification $\mathcal{S}$ are represented in the CSCFG associated to $\mathcal{S}$.

Lemma 1. *Let $\mathcal{S}$ be a CSP specification, $\mathcal{D} = s_0 \longrightarrow \ldots \longrightarrow s_{n+1}$, $n \geqslant 0$, a* derivation *of $\mathcal{S}$ performed with the instrumented semantics, where $s_0 = (rhs(\mathtt{MAIN})_\alpha, \varnothing)$ and $s_{n+1} = (P_\varphi, \omega)$, and $\mathcal{G} = (N, E_c, E_l, E_s)$ the CSCFG associated with $\mathcal{S}$. Then, $\forall \gamma \in \omega :\ \exists\ \pi = n_1 \mapsto^* n_k \in E_c$, $n_1, n_k \in N$, $k \geqslant 1$, with $l(n_1) = \alpha$ and $l(n_k) = \gamma$.*

This lemma ensures that all derivations are represented in the CSCFG with a path; but, of course, because it is a static representation of any possible execution, the CSCFG also introduces a source of imprecision. This imprecision happens when loop arcs are introduced in a CSCFG, because a loop arc summarizes the rest of a computation with a single collection of nodes, and this collection could mix synchronizations of different iterations. However, note that all process calls of the specification are unfolded and represented with an exclusive collection of nodes, and loop arcs are only introduced if the same call is repeated again. This produces a high level of precision for slicing algorithms.

Because Definition 8 is a declarative definition, it is not very useful for implementors; and hence, we also provide a constructive method that is the basis of our implementation. In particular, the CSCFG can be constructed starting from `MAIN`, and connecting each

process call to a subgraph that contains the right-hand side of the called process. Each right-hand side is a new subgraph except if a loop is detected. This process is described in Algorithm 1.

Algorithm 1 Computing the CSCFG

Input: A specification $\mathcal{S}$ with initial process `MAIN`
Output: The CSCFG $\mathcal{G}$ of $\mathcal{S}$
Begin
Pending={`MAIN`}
while *Pending* $\neq \emptyset$ **do**
(1) $(N', E'_c, E_l, n_{first}, Last) = buildGraph(rhs(P), \emptyset)$ where $P \in Pending$
(2) $N = N' \cup \{n_{start}, n_{end}\}$ where n_{start}, n_{end} are fresh,
$l(n_{start}) =$ "$start\ (P, 0)$" and $l(n_{end}) =$ "$end\ (P, 0)$"
(3) $E_c = E'_c \cup \{n_{start} \mapsto n_{first}\} \cup \{n_{last} \mapsto n_{end} \mid n_{last} \in Last\}$
(4) $Pending = \{P' \in \mathcal{P}roc(\mathcal{S}) \mid \nexists n \in N : lit(l(n)) = P'\}$
E_s is obtained following the technique from [23]
return $\mathcal{G} = (N, E_c, E_l, E_s)$
End

Function $buildGraph(P, Ctx) = (N, E_c, E_l, n_{first}, Last)$ where:

- **((Parameterized) Process call)**. If $P = X_\alpha$ and $\alpha \in \mathcal{C}alls(\mathcal{S})$ then
  - if $\exists n_{ctx} \in Ctx$ such that $l(n_{ctx}) =$ "$start\ \alpha$" then

$$N = \{n_\alpha\},\quad E_c = \emptyset,\quad E_l = \{(n_\alpha \rightsquigarrow n_{ctx})\},\quad n_{first} = n_\alpha \text{ and } Last = \emptyset$$

  - else

$$N = N_1 \cup \{n_\alpha, n_{start}, n_{end}\},\quad E_c = E_{c1} \cup E_{c2},\quad E_l = E_{l1},\quad n_{first} = n_\alpha \text{ and } Last = \{n_{end}\}$$

  where

$n_\alpha, n_{start}, n_{end}$ are $fresh\ \wedge\ l(n_\alpha) = \alpha\ \wedge\ l(n_{start}) =$ "$start\ \alpha$" $\wedge$
$l(n_{end}) =$ "$end\ \alpha$" $\wedge\ X = Q \in \mathcal{S}\ \wedge$
$(N_1, E_{c1}, E_{l1}, n_{first1}, Last_1) = buildGraph(Q, Ctx \cup \{n_{start}\})\ \wedge$
$E_{c2} = \{(n_\alpha \mapsto n_{start}), (n_{start} \mapsto n_{first1})\} \cup \{(n_{last} \mapsto n_{end}) \mid n_{last} \in Last_1\}$.

- **(Prefixing)**. If $P = X_\alpha \rightarrow_\beta Q$ and $X \in \{a, a?v, a!v\}$ then

$$N = N_1 \cup \{n_\alpha, n_\beta\},\quad E_c = E_{c1} \cup \{(n_\alpha \mapsto n_\beta), (n_\beta \mapsto n_{first1})\},\quad E_l = E_{l1},\quad n_{first} = n_\alpha \text{ and } Last = Last_1$$

where

$$n_\alpha, n_\beta \text{ are } fresh \quad \wedge \quad l(n_\alpha) = \alpha \quad \wedge \quad l(n_\beta) = \beta \quad \wedge$$
$$(N_1, E_{c1}, E_{l1}, n_{first1}, Last_1) = buildGraph(Q, Ctx).$$

- **(Choice and parallelism)**. If $P = Q\ X_\alpha\ R$ and $X \in \{\sqcap, \Box, \nleftrightarrow, |||, ||\}$ then

$$N = N_1 \cup N_2 \cup \{n_\alpha\},$$
$$E_c = E_{c1} \cup E_{c2} \cup \{(n_\alpha \mapsto n_{first1}), (n_\alpha \mapsto n_{first2})\},$$
$$E_l = E_{l1} \cup E_{l2},$$
$$n_{first} = n_\alpha \text{ and}$$

$$Last = \begin{cases} Last_1 \cup Last_2 & \text{if } X \in \{\sqcap, \Box, \nleftrightarrow\} \vee \\ & \quad (Last_1 \neq \emptyset \wedge Last_2 \neq \emptyset) \\ \emptyset & \text{if } X \in \{|||, ||\} \wedge \\ & \quad (Last_1 = \emptyset \vee Last_2 = \emptyset) \end{cases}$$

where

$$n_\alpha \text{ is } fresh \quad \wedge \quad l(n_\alpha) = \alpha \quad \wedge$$
$$(N_1, E_{c1}, E_{l1}, n_{first1}, Last_1) = buildGraph(Q, Ctx) \quad \wedge$$
$$(N_2, E_{c2}, E_{l2}, n_{first2}, Last_2) = buildGraph(R, Ctx).$$

- **(Sequential composition)**. If $P = Q\ ;_\alpha\ R$ then

$$N = N_1 \cup N_2 \cup \{n_\alpha\},$$
$$E_c = E_{c1} \cup E_{c2} \cup E_{c3} \cup \{(n_\alpha \mapsto n_{first2})\},$$
$$E_l = E_{l1} \cup E_{l2},$$
$$n_{first} = n_{first1} \text{ and } Last = Last_2$$

where

$$n_\alpha \text{ is } fresh \quad \wedge \quad l(n_\alpha) = \ \alpha \quad \wedge$$
$$(N_1, E_{c1}, E_{l1}, n_{first1}, Last_1) = buildGraph(Q, Ctx) \quad \wedge$$
$$(N_2, E_{c2}, E_{l2}, n_{first2}, Last_2) = buildGraph(R, Ctx) \quad \wedge$$
$$E_{c3} = \{(n_{last} \mapsto n_\alpha) \mid n_{last} \in Last_1\}.$$

- **(Hiding and renaming)**. If $P = Q\ X_\alpha$ and $X \in \{\backslash, [\![\,]\!]\}$ then

$$N = N_1 \cup \{n_\alpha, n_{end}\},$$
$$E_c = E_{c1} \cup E_{c2} \cup \{(n_\alpha \mapsto n_{first1})\},$$
$$E_l = E_{l1},$$
$$n_{first} = n_\alpha \text{ and } Last = \{n_{end}\}$$

where

$$n_\alpha, n_{end} \text{ are } fresh \quad \wedge \quad l(n_\alpha) = \alpha \quad \wedge \quad l(n_{end}) = \text{“}end\ \alpha\text{”} \quad \wedge$$
$$(N_1, E_{c1}, E_{l1}, n_{first1}, Last_1) = buildGraph(Q, Ctx) \quad \wedge$$
$$E_{c2} = \{(n_{last} \mapsto n_{end}) \mid n_{last} \in Last_1\}.$$

- **(SKIP and STOP)**. If $P = X_\alpha$ and $X \in \{SKIP, STOP\}$ then

$$N = \{n_\alpha\},$$
$$E_c = \emptyset,$$
$$E_l = \emptyset,$$
$$n_{first} = n_\alpha \text{ and}$$

$$Last = \begin{cases} \{n_\alpha\} & \text{if } X = SKIP \\ \varnothing & \text{if } X = STOP \end{cases}$$

where

$$n_\alpha \text{ is } fresh \quad \wedge \quad l(n_\alpha) = \alpha.$$

The following Lemma ensures that the graph produced by Algorithm 1 is a CSCFG.

Lemma 2. *Let $\mathcal{S}$ be a CSP specification. Then, the execution of Algorithm 1 with $\mathcal{S}$ produces a graph $\mathcal{G}$ that is the CSCFG associated with $\mathcal{S}$ according to Definition 8.*

For slicing purposes, the CSCFG is interesting because we can use the edges to determine if a node must be executed or not before another node, thanks to the following properties:

- if $n \mapsto n' \in E_c$ then n must be executed before n' in all executions.
- if $n \rightsquigarrow n' \in E_l$ then n' must be executed before n in all executions.
- if $n \leftrightarrow n' \in E_s$ then n and n' are executed at the same time in all executions.

While the third property is obvious and it follows from the semantics of synchronized parallelism (concretely, from rule (Synchronized Parallelism 3)), the other two properties require proof. The second property follows trivially from the first property and Definition 8, because loop edges only connect a process call node to a node already repeated in the computation. The first property corresponds to Lemma 3.

Lemma 3. *Let $\mathcal{S}$ be a CSP specification and let $\mathcal{G} = (N, E_c, E_l, E_s)$ be the CSCFG associated with $\mathcal{S}$ according to Definition 8. If $n \mapsto n' \in E_c$ then n must be executed before n' in all executions.*

Thanks to the fact that loops are unfolded only once, the CSCFG ensures that all the specification positions inside the loops are in the graph and can be collected by slicing algorithms. For slicing purposes, this representation also ensures that every possibly executed part of the specification belongs to the CSCFG because only loops (i.e., repeated nodes) are missing.

Example 6. *Consider the specification of Example 3 and its associated CSCFG shown in Figure 6(b). If we select the node labeled* `(P1,alight)` *and traverse the CSCFG backwards in order to identify the nodes on which this node depends, we only get the nodes of the graph colored in gray. This particular slice is optimal and much smaller than the slice obtained when we select the same node* `(P1,alight)` *in the SCFG (see Figure 6(a)).*

The CSCFG provides a different representation for each context in which a process call is made. This can be seen in Figure 6(b) where process `BUS` appears twice to account for the two contexts in which it is called. In particular, in the CSCFG we have a fresh node to represent each different process call, and two nodes point to the same process if and only if they are the same call (they are labeled with the same specification position) and they belong to the same loop. This property ensures that the CSCFG is finite.

Lemma 4. *(Finiteness) Given a specification $\mathcal{S}$, its associated CSCFG is finite.*

Example 7. *The specification in Figure 7 makes clear the difference between the SCFG and the CSCFG. While the SCFG only uses one representation for the process* `P` *(there is only one* `start P`*), the CSCFG uses four different representations because* `P` *could be executed in four different contexts. Note that due to the infinite loops, some parts of the graph are not reachable from* `start MAIN`*; i.e., there is no possible control flow to* `end MAIN`*.*

```
MAIN = P ; P

P = Q

Q = P
```

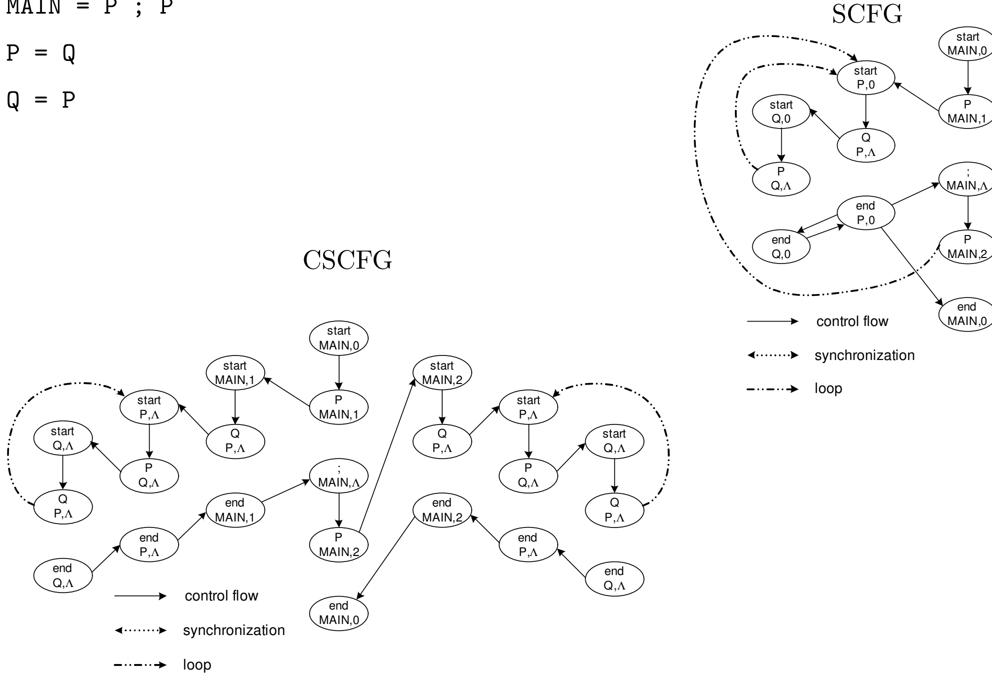


Figure 7: SCFG and CSCFG representing an infinite computation

4. Static Slicing of CSP Specifications

We want to perform two kinds of analysis. Given a point in the specification, we want, on the one hand, to determine what parts of the specification MUST be executed before (MEB) it (in every possible execution); and, on the other hand, we want to determine what parts of the specification COULD be executed before (CEB) it (in any possible execution). Both analyses are closely related but they must be computed differently. While MEB is mainly based on backward slicing, CEB is mainly based on forward slicing to explore what could be executed in parallel processes.

We can now formally define our notion of slicing criterion.

Definition 9. (Slicing Criterion) Given a specification $\mathcal{S}$, a *slicing criterion* is a specification position $\mathcal{C} \in \mathcal{P}os(\mathcal{S})$.

Clearly, the slicing criterion points to a set of nodes in the CSCFG, because the same specification position can happen in different contexts and, thus, it is represented in the CSCFG with different nodes. As an example, consider the slicing criterion (BUS,alight) for the specification in Example 3, and observe in its CSCFG in Figure 6(b) that two different nodes are identified by the slicing criterion.

This means that a slicing criterion $\mathcal{C}$ is used to produce a slice with respect to *all* possible executions of $\mathcal{C}$. We use the function $nodes(\mathcal{C})$ to refer to all the nodes in the CSCFG identified by the slicing criterion $\mathcal{C}$. Formally, given a CSCFG $\mathcal{G} = (N, E_c, E_l, E_s)$,

$$nodes(\mathcal{C}) = \{n \in N \mid l(n) = \mathcal{C} \ \wedge \texttt{MAIN} \mapsto^* n \ \wedge \ \nexists\, n' \in N \mid l(n') = \mathcal{C} \text{ and } n' \mapsto^* n\}$$

Note that the slicing criterion could point to nodes that are not reachable from MAIN such as dead code (see, e.g., Figure 7). Therefore, we force $nodes(\mathcal{C})$ to exclude these nodes so that only feasible computations (starting from MAIN) are considered. Moreover, the slicing criterion could also point to different nodes that represent the same specification position that is executed many times in a (sub)computation (see, e.g., specification position (P,Λ) in the CSCFG of Figure 7). Thus, we only select the first occurrence of this specification position in the computation.

Given a slicing criterion (M, w), we use the CSCFG to approximate MEB and CEB. Computing correct slices is known as an undecidable problem even in the sequential setting (see, e.g., [28]). Therefore, our MEB and CEB analyses are an over-approximation. In this section we introduce lemmas to ensure the completeness of the analyses.

Regarding the MEB analysis, one could think that a simple backwards traversal of the graph from $nodes(\mathcal{C})$ would produce a correct slice. Nevertheless, this would produce a rather imprecise slice because this would include both branches of the choices in the path from MAIN to $\mathcal{C}$ even if they do not need to be executed before $\mathcal{C}$ (consider for instance the process ((a→SKIP)□(b→SKIP));P and the slicing criterion P). The union of paths from MAIN to $nodes(\mathcal{C})$ is not a solution, either, because it would be too imprecise by including in the slice parts of code that are executed before the slicing criterion only in some executions. For instance, in the process (b→a→SKIP)□(c→a→SKIP), c belongs to one of the paths to a, but it must be executed before a or not depending on the choice. The intersection of paths is not a solution, either, as it can be seen in the process a→((b→SKIP)||(c→SKIP));P where b must be executed before P, but it does not belong to all the paths from MAIN to P.

Before we introduce an algorithm to compute MEB, we need to formally define the notion of MEB slice.

Definition 10. (MEB Slice) Given a specification $\mathcal{S}$ with an associated CSCFG $\mathcal{G} = (N, E_c, E_l, E_s)$, and a slicing criterion $\mathcal{C}$ for $\mathcal{S}$; the *MEB slice* of $\mathcal{S}$ with respect to $\mathcal{C}$ is a subset $\mathcal{P}'$ of $\mathcal{P}os(\mathcal{S})$ such that $\mathcal{P}' = \bigcap\{\omega \mid (\texttt{MAIN}, \varnothing) \rightarrow^* (P, \omega) \rightarrow (P', \omega') \wedge \mathcal{C} \notin \omega \wedge \mathcal{C} \in \omega'\}$.

Algorithm 2 can be used to compute the MEB analysis. It basically computes for each node in $nodes(\mathcal{C})$ a set containing the part of the specification that must be executed before it. Then, it returns MEB as the intersection of all these sets. Each set is computed with function *buildMeb*, which is an iterative process that takes a node and performs the following actions:

1. It starts with an initial set of nodes computed in (1) by collecting those nodes that were executed just before the initial node (i.e., they are connected to it or to a node synchronized with it with a control arc).
2. The initial set *Meb* is the backwards traversal of the CSCFG from the initial set following control arcs (2).
3. Those nodes that could not be executed before the initial node are added to a blacklist (3) and (4). The nodes in the blacklist are discarded because they are either a successor of the nodes in the slicing criterion (and thus they are executed always after it), or they are executed in a branch of a choice that cannot lead to the slicing criterion. Note that the blacklist in sentence (4) is computed by iteratively collecting all the nodes that are a (control) successor of the nodes in the previous blacklist (initially the slicing criterion); and it also adds to the blacklist those nodes that are only synchronized with nodes in the blacklist.
4. A set of *pending* nodes that should be considered is computed in (5). This set contains nodes that are synchronized with the nodes in *Meb* (thus they are executed at the same time). Therefore, synchronizations are followed in order to reach new nodes that must be executed before the slicing criterion (7) and (8). These steps are repeated until no new nodes are reached. This is controlled with the set *pending* (6) and (9).

The algorithm always terminates as stated in the following lemma.

Theorem 8 (Termination of MEB)**.** *The MEB analysis performed by Algorithm 2 terminates.*

Theorem 9 (Completeness of MEB)**.** *Let $\mathcal{S}$ be a specification, $\mathcal{C}$ a slicing criterion for $\mathcal{S}$, and let $\mathcal{MEB}$ be the MEB slice of $\mathcal{S}$ with respect to $\mathcal{C}$. Then, $\mathcal{MEB} \subseteq MEB(\mathcal{S}, \mathcal{C})$.*

The CEB analysis computes the set of nodes in the CSCFG that could be executed before a given node n. This means that all those nodes that must be executed before n are included, but also those nodes that are executed before n in some executions, and they are not in other executions (e.g., due to non-synchronized parallelism). Formally,

Definition 11. (CEB Slice) Given a specification $\mathcal{S}$ with an associated CSCFG $\mathcal{G} = (N, E_c, E_l, E_s)$, and a slicing criterion $\mathcal{C}$ for $\mathcal{S}$; the *CEB slice* of $\mathcal{S}$ with respect to $\mathcal{C}$ is a subset $\mathcal{P}'$ of $\mathcal{P}os(\mathcal{S})$ such that $\mathcal{P}' = \bigcup\{\omega \mid (\texttt{MAIN}, \varnothing) \rightarrow^* (P, \omega) \rightarrow (P', \omega') \wedge \mathcal{C} \notin \omega \wedge \mathcal{C} \in \omega'\}$.

Therefore, $MEB(\mathcal{S}, \mathcal{C}) \subseteq CEB(\mathcal{S}, \mathcal{C})$.

The graph $CEB(\mathcal{S}, \mathcal{C})$ can be computed with Algorithm 3 that, roughly, traverses the CSCFG forwards following all the paths that could be executed in parallel to nodes in $MEB(\mathcal{S}, \mathcal{C})$. In particular, the algorithm computes for each node in $nodes(\mathcal{C})$ a set containing the part of the specification that could be executed before it. Then, it returns CEB as the union of all these sets. Each set is computed with function *buildCeb*, which proceeds as follows:

1. In sentence (1), it initializes the set *Ceb* with function *buildMeb* (trivially, all those specification positions that must be executed before a node n, could be executed before it).

Algorithm 2 Computing the MEB set

Input: A CSCFG (N, E_c, E_l, E_s) of a specification $\mathcal{S}$ and a slicing criterion $\mathcal{C}$
Output: A slice of $\mathcal{S}$

Function $\mathit{buildMeb}(n)$:=
(1) $\mathit{init} := \{n' \mid (n' \mapsto o) \in E_c\}$ where $o \in \{n\} \cup \{o' \mid (o' \leftrightsquigarrow n) \in E_s\}$
(2) $\mathit{Meb} := \{o \in (\texttt{MAIN} \mapsto^* m) \mid m \in \mathit{init}\}$
(3) $\mathit{blacklist} := \{n\} \cup \{p \in N \backslash \mathit{Meb} \mid (o \mapsto p) \in E_c$ with $lit(l(o)) \in \{\sqcap, \Box, |||\}$, $o \in \mathit{Meb}$ and $\nexists\ q \in \mathit{Meb}$ such that q is reachable from p following control or loop arcs$\}$
repeat
(4) $\mathit{blacklist} := \mathit{blacklist} \cup \{p \in N \mid o \mapsto^* p$ with $o \in \mathit{blacklist}\}$ $\cup \{p \in N \mid (o \leftrightsquigarrow p) \in E_s$ with $o \in \mathit{blacklist}$ and $\nexists (p \leftrightsquigarrow p') \in E_s$ with $p' \notin \mathit{blacklist}\}$
until a fix point is reached
(5) $\mathit{pending} := \{q \in N \backslash (\mathit{blacklist} \cup \mathit{Meb}) \mid (q \leftrightsquigarrow r) \in E_s$ with $r \in \mathit{Meb}$ or $((q \rightsquigarrow r) \in E_l$ and $\forall\ s \in (r \mapsto^* q)\ .\ (\exists (s \leftrightsquigarrow t) \in E_s$ with $t \in \mathit{Meb}$ or $\nexists (s \leftrightsquigarrow t) \in E_s))\}$
(6) while $\exists\ m \in \mathit{pending}$ do
(7) $\mathit{Meb} := \mathit{Meb} \cup \{m\} \cup \{o \in N \backslash \mathit{Meb} \mid (p \mapsto^+ o \mapsto^* m)$ with $p \in \mathit{Meb}\}$
(8) $\mathit{sync} := \{q \in N \backslash (\mathit{blacklist} \cup \mathit{Meb} \cup \mathit{pending}) \mid (q \leftrightsquigarrow r) \in E_s$ with $r \in \mathit{Meb}$ or $((q \rightsquigarrow r) \in E_l$ and $\forall\ s \in (r \mapsto^* q)\ .\ (\exists (s \leftrightsquigarrow t) \in E_s$ with $t \in \mathit{Meb}$ or $\nexists (s \leftrightsquigarrow t) \in E_s))\}$
(9) $\mathit{pending} := (\mathit{pending} \backslash \mathit{Meb}) \cup \mathit{sync}$
(10) return Meb

Return: $MEB(\mathcal{S}, \mathcal{C}) = \bigcap_{n \in nodes(\mathcal{C})} \{l(n') \mid n' \in \mathit{buildMeb}(n)\}$

2. In sentence (2), it initializes the set *loopnodes*. This set represents the nodes that belong to a loop in the computation executed before the slicing criterion was reached. For instance, in the process `A=(a→A)`□`(b→SKIP)` the left branch of the choice is a loop that could be executed several times before the slicing criterion, say `b`, was executed. Initially, this set contains the first node in a branch of a choice operator that does not belong to *Ceb* but can reach *Ceb* through a loop arc.
3. The set *loopnodes* is computed in the first loop of the algorithm, sentences (4) to (10) and they are finally added to the slice (i.e., *Ceb*). In particular, sentence (11) checks that the whole loop could be executed before the slicing criterion. If some sentence of the loop could not be executed before (e.g., because it is synchronized with an event that must occur after the slicing criterion), then the loop is discarded and not included in the slice.
4. The second loop of the algorithm, sentences (12) to (18), is used to collect all those nodes that could be executed in parallel to the nodes in the slice (in *Ceb*). In particular, it traverses branches executed in parallel to nodes in *Ceb* until a node that could not be executed before the slicing criterion is found. For instance,

consider the process `A=(a→b→SKIP)`$||_{\{\mathtt{b}\}}$`(c→b→SKIP)`; and let us assume that the slicing criterion is `c`. Similarly to the first loop of the algorithm, the second loop traverses the left branch of the parallelism operator forwards until an event that could not be executed before the slicing criterion is found (in this example, `b`). Therefore, `a→` would be included in the slice.

Algorithm 3 Computing the CEB set

Input: A CSCFG (N, E_c, E_l, E_s) of a specification $\mathcal{S}$ and a slicing criterion $\mathcal{C}$
Output: A slice of $\mathcal{S}$

Function $buildCeb(n) :=$
(1) $Ceb := buildMeb(n)$
(2) $loopnodes := \{p \mid n_1 \mapsto p \mapsto^* n_2 \rightsquigarrow n_3$ with $n_1 \in choices(Ceb)$, $p, n_2 \notin Ceb$ and $n_3 \in Ceb\}$
(3) $candidates := \emptyset$
repeat
(4) if $\exists (m \mapsto m') \in E_c$ with $m \in loopnodes$ and $m' \notin loopnodes$
(5) then if $\exists (m' \leftrightarrow m'') \in E_s$ with $m'' \in Ceb$ or $\nexists (m' \leftrightarrow m'') \in E_s)$
(6) then $loopnodes := loopnodes \cup \{m'\}$
(7) else $candidates := candidates \cup \{m'\}$
(8) if $\exists (m \leftrightarrow m') \in E_s$ and $m, m' \in candidates$
(9) then $loopnodes := loopnodes \cup \{m, m'\}$
(10) $candidates := candidates \backslash \{m, m'\}$
until a fix point is reached
(11) $Ceb := Ceb \cup \{p \in loopnodes \mid \forall\ o \in loopnodes, p \mapsto^* o \rightsquigarrow q$ with $q \in Ceb\}$
(12) $pending := \{m \in N \backslash (Ceb \cup \{n\}) \mid (m' \mapsto m) \in E_c$ and $m' \in Ceb \backslash choices(Ceb)\}$
repeat
(13) if $\exists\ m \in pending \mid (m \leftrightarrow m') \notin E_s$ or $((m \leftrightarrow m') \in E_s$ and $m' \in Ceb)$
(14) then $Ceb := Ceb \cup \{m\}$
(15) $pending := (pending \backslash \{m\}) \cup \{m'' \mid (m \mapsto m'') \in E_c$ and $m'' \notin Ceb\}$
(16) else if $\exists\ m \in pending$ and $(m \leftrightarrow m') \in E_s$ with $m' \in pending$
(17) then $Ceb := Ceb \cup \{m, m'\}$
(18) $pending := (pending \backslash \{m, m'\}) \cup \{p \mid (o \mapsto p) \in E_c$ and $p \notin Ceb$, with $o \in \{m, m'\}\}$
until a fix point is reached
(19) return Ceb

Return: $CEB(\mathcal{S}, \mathcal{C}) = \bigcup_{n \in nodes(\mathcal{C})} \{l(n') \mid n' \in buildCeb(n)\}$

The algorithms presented can extract a slice from any specification formed with the syntax of Figure 1. However, note that only two operators have a special treatment in the algorithms: choices (because they introduce alternative computations) and synchronized parallelism constructs (because they introduce synchronization). Other operators such as prefixing, interleaving or sequential composition are only taken into account in the

CSCFG construction phase; and they can be treated similarly in the algorithm (i.e., they are traversed forwards or backwards by the algorithm when exploring computations).

Theorem 10 (Termination of CEB)**.** *The CEB analysis performed by Algorithm 3 terminates.*

Theorem 11 (Completeness of CEB)**.** *Let $\mathcal{S}$ be a specification, $\mathcal{C}$ a slicing criterion for $\mathcal{S}$, and let $\mathcal{CEB}$ be the CEB slice of $\mathcal{S}$ with respect to $\mathcal{C}$. Then, $\mathcal{CEB} \subseteq CEB(\mathcal{S},\mathcal{C})$.*

5. Implementation

We have implemented the MEB and CEB analyses and the algorithms to build the CSCFG for ProB. ProB [16] is an animator for the B-Method which also supports other languages such as CSP [4, 17]. ProB has been implemented in Prolog and it is publicly available at `http://www.stups.uni-duesseldorf.de/ProB`.

Our tool is called SOC (which stands for *Slicing Of CSP*) and it is currently integrated, distributed and maintained for Mac, Linux and Windows since the 1.3 release of ProB. In SOC, the slicing process is completely automatic. Once the user has loaded a CSP specification, she can select (with the mouse) the event, operator or process call she is interested in. Obviously, this simple action is enough to define a slicing criterion because the tool can automatically determine the process and the source position of interest. Then, the tool internally generates an internal data structure (the CSCFG) that represents all possible computations, and uses the MEB and CEB algorithms to construct the slices. The result is shown to the user by highlighting the part of the specification that must (respectively could) be executed before the specified event. Figure 8 shows a screenshot of the tool showing a slice of the specification in Example 1. SOC also includes a transformation to convert slices into executable programs. This allows us to use SOC for program specialization. The specialized versions produced can be directly executed in ProB.

5.1. Architecture of SOC

SOC has been implemented in Prolog and it has been integrated in ProB. Therefore, SOC can take advantage of ProB's graphical features to show slices to the user. In order to be able to color parts of the code, it has been necessary to implement the source code positions detection in such a way that ProB can color every subexpression that is sliced by SOC.

Figure 9 summarizes the internal architecture of SOC. Note that both the graph compaction module and the slicing module take a CSCFG as input, and hence, they are independent of CSP. Apart from the interface module for the communication with ProB, SOC has three main modules that we describe in the following:

Graph Generation

The first task of the slicer is to build a CSCFG. The module that generates the CSCFG from the source program is the only module that is CSP dependent. This means that SOC could be used in other languages by only changing the graph generation module.

Nodes and control and loop arcs are built following Definition 8. For synchronization edges we use an algorithm based on the approach by Naumovich et al. [23]. For efficiency

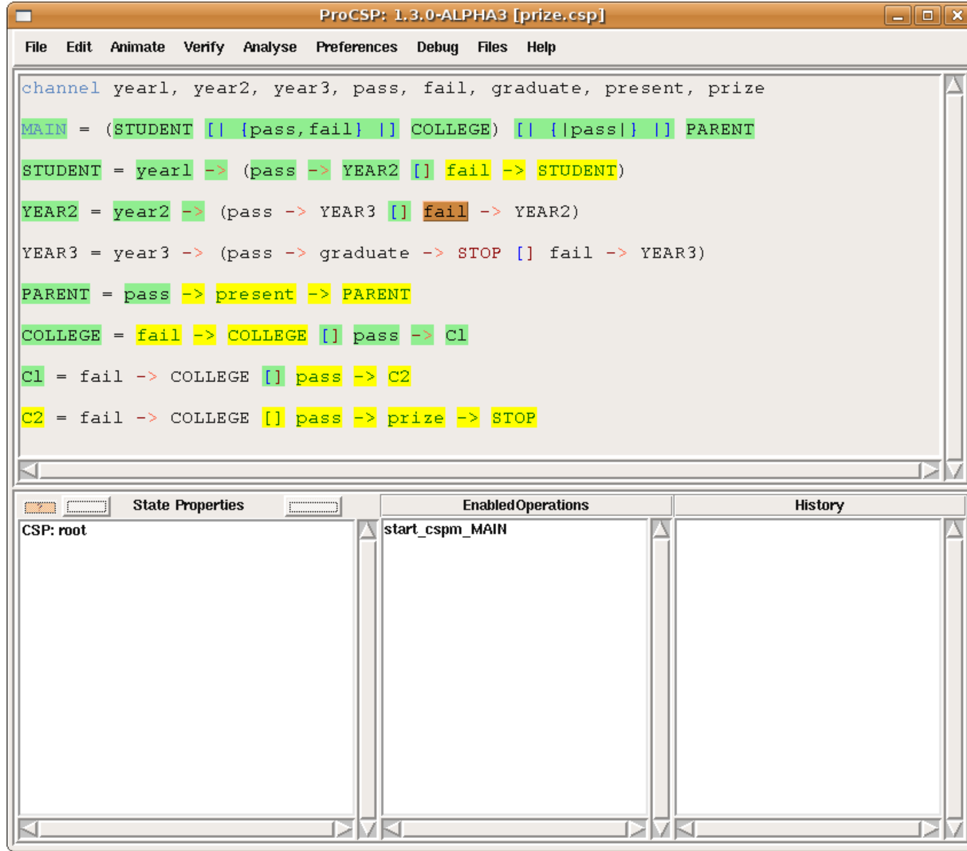


Figure 8: Slice of a CSP specification produced by SOC

reasons, the implementation of the CSCFG makes some simplifications that reduce the size of the graph. For instance, "*start*" and "*end*" nodes are not present in the graph. Another simplification to reduce the size of the graph is graph compaction (described below).

We have implemented two versions of this module. The first version has the objective of producing a precise analysis. For this purpose, the notion of context described in Definition 6 is used together with the first property of Definition 8. Recall that this property uses the context to introduce loop arcs in the graph whenever a specification position is repeated in a loop. However, this notion of context can produce big CSCFGs with some examples. This implies more memory usage and more time to compute the graphs and the slices. In such cases, the user could be interested in producing the CSCFG as fast as possible; for instance, when the analysis is used as a preprocessing stage of another analysis. Therefore, we have produced a lightweight version to produce a fast analysis when necessary. This second version uses a relaxed notion of context that allows the CSCFG to cut more branches of the graph with loop arcs. The fast analysis replaces property one in Definition 8 by

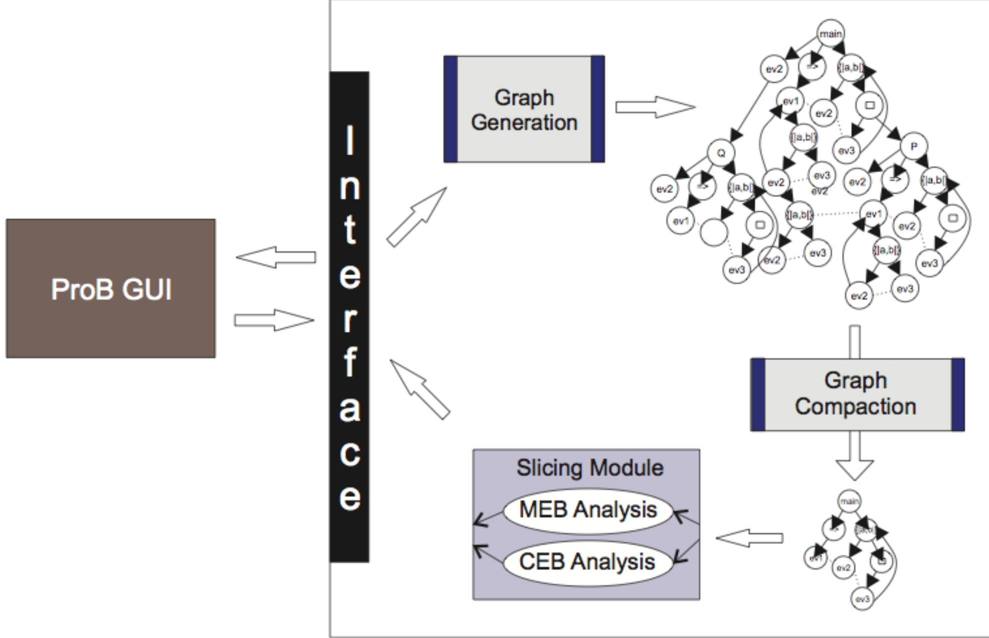


Figure 9: Slicer's Architecture

- There is a special set of *loop arcs* (E_l) denoted with $\rightsquigarrow$. $(n_1 \rightsquigarrow n_2) \in E_l$ iff $l(n_1) \in \mathcal{C}alls(\mathcal{S}) \wedge l(n_2) =$ "*start s*" $\wedge\ lit(l(n_1)) = lit(s) \wedge n_2 \in \mathcal{C}on(n_1)$.

which skips the restriction that the specification position of n_1 must be repeated. Therefore, while the *precise* context only introduces a loop arc in the CSCFG when the same specification position is repeated in a branch, the *fast* context introduces a loop arc when the same process call is repeated, even if the specification position of the call is different.

Example 12. *Consider again the CSCFG in Figure 7. This CSCFG corresponds to the precise context, and thus loop arcs are only used when the same specification position is repeated. In contrast, the CSCFG constructed using the fast context uses loop arcs whenever the same process call is repeated (i.e., the literal). It is depicted in Figure 10.*

Both analyses have been compared with several benchmarks. The results are presented in Section 5.2.

Graph Compaction

For the sake of clarity, the definition of CSCFG proposed does not take into account efficiency. In particular, it includes several nodes that are unnecessary from an implementation point of view. Therefore, we have implemented a module that reduces the size of the CSCFG by removing superfluous nodes and by joining together those nodes that form paths that the slicing algorithms must traverse in all cases. This compaction not only reduces the size of the stored CSCFG, but it also speeds up the slicing process due to the reduced number of nodes to be processed.

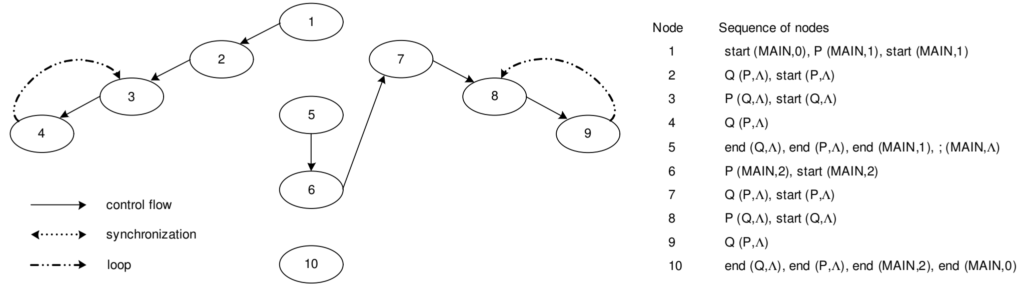


Figure 10: CSCFG of the specification in Figure 7 using the fast context.

For instance, the graph of Figure 11 is the compacted version of the CSCFG in Figure 6(b). Here, e.g., node `2` accounts for the sequence of nodes `BUS` and `start BUS`. The compacted version is a very convenient representation because the reduced data structure speeds up the graph traversal process. In practice, the graph compaction phase reduces the size of the graph up to 40% on average.

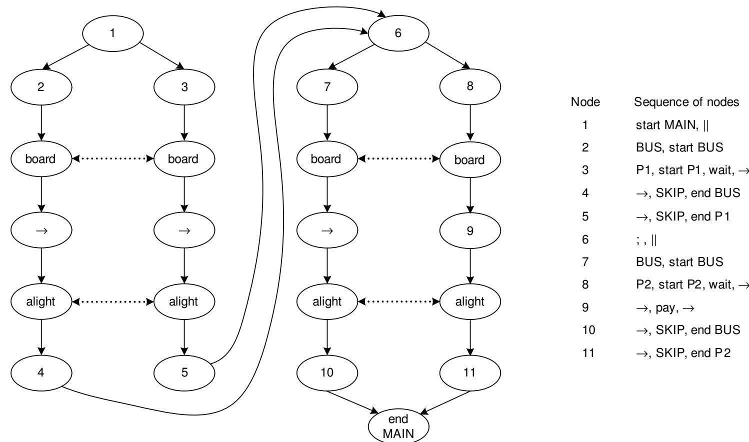


Figure 11: Compacted version of the CSCFG in Figure 6(b)

Slicing Module

This is the main module of the tool. It is further composed of two submodules that implement the algorithms to perform the MEB and CEB analyses on the compacted CSCFGs. This module extracts two subgraphs from the compacted CSCFG using both MEB and CEB. Then, it extracts from the subgraphs the part of the source code which forms the slice. This information can be extracted directly from the graph because its nodes are labeled with the specification positions to be highlighted. If the user has selected to produce an executable slice, then the slice is further transformed to become executable (it mainly fills gaps in the produced slice in order to respect the syntax of the language). The final result is then returned to ProB in such a way that ProB can either highlight the final slice or save a new CSP executable specification in a file.

Table 1: Benchmark time results for the FAST and PRECISE CONTEXT

(a) Benchmark time results for the FAST CONTEXT

Benchmark	CSCFG	MEB	CEB	Total
ATM.csp	805 ms.	36 ms.	67 ms.	908 ms.
RobotControl.csp	277 ms.	39 ms.	21 ms.	337 ms.
Buses.csp	29 ms.	2 ms.	1 ms.	32 ms.
Prize.csp	55 ms.	35 ms.	10 ms.	100 ms.
Phils.csp	72 ms.	12 ms.	4 ms.	88 ms.
TrafficLights.csp	103 ms.	20 ms.	12 ms.	135 ms.
Processors.csp	10 ms.	4 ms.	2 ms.	16 ms.
ComplexSync.csp	212 ms.	264 ms.	38 ms.	514 ms.
Computers.csp	23 ms.	6 ms.	1 ms.	30 ms.
Highways.csp	11452 ms.	100 ms.	30 ms.	11582 ms.

(b) Benchmark time results for the PRECISE CONTEXT

Benchmark	CSCFG	MEB	CEB	Total
ATM.csp	10632 ms.	190 ms.	272 ms.	11094 ms.
RobotControl.csp	2603 ms.	413 ms.	169 ms.	3185 ms.
Buses.csp	25 ms.	1 ms.	0 ms.	26 ms.
Prize.csp	352 ms.	317 ms.	79 ms.	748 ms.
Phils.csp	96 ms.	12 ms.	8 ms.	116 ms.
TrafficLights.csp	2109 ms.	1678 ms.	416 ms.	4203 ms.
Processors.csp	15 ms.	2 ms.	5 ms.	22 ms.
ComplexSync.csp	23912 ms.	552 ms.	174 ms.	24638 ms.
Computers.csp	51 ms.	4 ms.	6 ms.	61 ms.
Highways.csp	58254 ms.	1846 ms.	2086 ms.	62186 ms.

5.2. Benchmarking the slicer

In order to measure the performance and the slicing capabilities of our tool, we conducted some experiments over the following benchmarks:

- `ATM.csp`. This specification represents an Automated Teller Machine. The slicing criterion is `(Menu,getmoney)`, i.e., we are interested in determining what parts of the specification must be executed before the menu option `getmoney` is chosen in the ATM.
- `RobotControl.csp`. This example describes a game in which four robots move in a maze. The slicing criterion is `(Referee,winner2)`, i.e., we want to know what parts of the system could be executed before the second robot wins.
- `Buses.csp`. This example describes a bus service with two buses running in parallel. The slicing criterion is `(BUS37, pay90)`, i.e., we are interested in determining what could and could not happen before the user payed at bus 37.

Table 2: Benchmark size results for the FAST and PRECISE CONTEXT

(a) Benchmark size results for the FAST CONTEXT

Benchmark	Ori_CSCFG	Com_CSCFG	(%)	MEB Slice	CEB Slice
ATM.csp	156 nodes	99 nodes	63.46 %	32 nodes	45 nodes
RobotControl.csp	337 nodes	121 nodes	35.91 %	22 nodes	109 nodes
Buses.csp	20 nodes	20 nodes	90.91 %	11 nodes	11 nodes
Prize.csp	70 nodes	52 nodes	74.29 %	25 nodes	42 nodes
Phils.csp	181 nodes	57 nodes	31.49 %	9 nodes	39 nodes
TrafficLights.csp	113 nodes	79 nodes	69.91 %	7 nodes	60 nodes
Processors.csp	30 nodes	15 nodes	50.00 %	8 nodes	9 nodes
ComplexSync.csp	103 nodes	69 nodes	66.99 %	37 nodes	69 nodes
Computers.csp	53 nodes	34 nodes	64.15 %	18 nodes	29 nodes
Highways.csp	103 nodes	62 nodes	60.19 %	41 nodes	48 nodes

(b) Benchmark size results for the PRECISE CONTEXT

Benchmark	Ori_CSCFG	Com_CSCFG	(%)	MEB Slice	CEB Slice
ATM.csp	267 nodes	165 nodes	61.8 %	52 nodes	59 nodes
RobotControl.csp	1139 nodes	393 nodes	34.5 %	58 nodes	369 nodes
Buses.csp	22 nodes	20 nodes	90.91 %	11 nodes	11 nodes
Prize.csp	248 nodes	178 nodes	71.77 %	15 nodes	47 nodes
Phils.csp	251 nodes	56 nodes	22.31 %	9 nodes	39 nodes
TrafficLights.csp	434 nodes	267 nodes	61.52 %	7 nodes	217 nodes
Processors.csp	37 nodes	19 nodes	51.35 %	8 nodes	14 nodes
ComplexSync.csp	196 nodes	131 nodes	66.84 %	18 nodes	96 nodes
Computers.csp	109 nodes	72 nodes	66.06 %	16 nodes	67 nodes
Highways.csp	503 nodes	275 nodes	54.67 %	47 nodes	273 nodes

- `Prize.csp`. This is the specification of Example 1. Here, the slicing criterion is `(YEAR2, fail)`, i.e., we are interested in determining what parts of the specification must be executed before the student fails in the second year.

- `Phils.csp`. This is a simple version of the dining philosophers problem. In this example, the slicing criterion is `(PHIL221, DropFork2)`, i.e., we want to know what happened before the second philosopher dropped the second fork.

- `TrafficLights.csp`. This specification defines two cars driving in parallel on different streets with traffic lights for cars controlling. The slicing criterion is `(STREET3,park)`, i.e., we are interested in producing an executable version of the specification in which we could simulate the executions where the second car parks on the third street.

- `Processors.csp`. This example describes a system that, once connected, receives data from two machines. The slicing criterion is `(MACH1,datreq)` to know what parts of the example must be executed before the first machine requests data.

- `ComplexSync.csp`. This specification defines five routers working in parallel. Router i can only send messages to router $i+1$. Each router can send a broadcast message to all routers. The slicing criterion is `(Process3,keep)`, i.e., we want to know what parts of the system could be executed before router 3 keeps a message.

- `Computers.csp`. This benchmark describes a system in which a user can surf internet and download files. The computer can check wether files are infected by virus. The slicing criterion is `(USER,consult_file)`, i.e., we are interested in determining what parts of the specification must be executed before the user consults a file.

- `Highways.csp`. This specification describes a net of spanish highways. The slicing criterion is `(HW6,Toledo)`, i.e., we want to determine what cities must be traversed in order to reach Toledo from the starting point.

All the source code and other information about the benchmarks can be found at

`http://www.dsic.upv.es/~jsilva/soc/examples`

For each benchmark, Table 1(a) and Table 1(b) summarize the time spent to generate the compacted CSCFG (this includes the generation plus the compaction phases), to produce the MEB and CEB slices (since CEB analysis uses MEB analysis, CEB's time corresponds only to the time spent after performing the MEB analysis), and the total time. Table 1(a) shows the results when using the fast context and Table 1(b) shows the results associated to the precise context. Clearly, the fast context achieves a significative time reduction. In these tables we can observe that Highways.csp needs more time even though the size of its associated CSCFG is similar to the other examples. Almost all the time needed to construct the CSCFG is used in computing the synchronizations. The high number of synchronizations performed in Highways.csp is the cause of its expensive cost.

Table 2(a) and Table 2(b) summarize the size of all objects participating in the slicing process for both the fast and the precise contexts respectively: Column `Ori_CSCFG` shows the size of the CSCFG of the original program. Observe that the precise context can increase the size of the CSCFG up to four times with respect to the fast context. Column `Com_CSCFG` shows the size of the compacted CSCFG. Column `(%)` shows the percentage of the compacted CSCFG' size with respect to the original CSCFG. Note that in some examples the reduction is almost 70% of the original size. Finally, columns `MEB Slice` and `CEB Slice` show respectively the size of the MEB and CEB CSCFG' slices. Clearly, CEB slices are always equal or greater than their MEB counterparts.

The CSCFG compaction technique seems to be useful. Experiments show that the size of the original specification is substantially reduced using this technique. The size of both MEB and CEB slices obviously depends on the slicing criterion selected. Table 2(a) and Table 2(b) compare both slices with respect to the same criterion but different contexts and, therefore, they give an idea of the difference between them.

SOC is open and publicly available. All the information related to the experiments, the source code of the benchmarks, the slicing criteria used, the source code of the tool and other material related to the project can be found at

`http://www.dsic.upv.es/~jsilva/soc`

6. Related Work

Program slicing has been already applied to concurrent programs of different programming paradigms, see e.g. [30, 29]. As a result, different graph representations have arisen to represent synchronization. The first proposal of a program slicing method for concurrent programs by Cheng [6] was later improved by Krinke [13, 14] and Nanda [21]. All these approaches are based on the so called *threaded control flow graph* and the *threaded program dependence graph*. Unfortunately, their approaches are not appropriate for slicing CSP, because their work is based on a different kind of synchronization. They use the following concept of *interference* to represent program synchronization.

Definition 12. (Interference) A node $S1$ is *interference* dependent on a node $S2$ if $S2$ defines a variable v, $S1$ uses the variable v and $S1$ and $S2$ execute in parallel.

In CSP, in contrast, a synchronization happens between two processes if the synchronized event is executed at the same time by both processes. In addition, both processes cannot proceed in their executions until they have synchronized. This is the key point that underpin our MEB and CEB analyses. This idea has been already exploited in the *concurrent control flow graph* [8] which allows us to model the phenomenon known as *fully-blocking* semantics where a process sending a message to other process is blocked until the other receives the message and vice versa. This is equivalent to our synchronization model. In these graphs, as in previous approaches (and in conventional program slicing in general), the slicing criterion is a variable in a point of interest, and the slice is formed by the sentences that *influence* this variable due to control and data dependences. For instance, consider the following program fragment:

(1) $read(x);$
(2) $print(x);$
(3) $if \;\; x > 0$
(4) $\qquad then \; y = x - 1;$
(5) $\qquad else \; y = \; 42;$
(6) $print(y);$
(7) $z = y;$

A slice with respect to $(7, z)$ would contain sentences (1), (3), (4) and (5); because z data depends on y, y data depends on x and (4) and (5) control depend on (3). Sentences (2) and (6) would be discarded because they are print statements and thus, they do not have an influence on z.

In contrast, in our technique, if we select (7) as the slicing criterion, we get sentences (1), (2), (3) and (6) as the MEB slice because these sentences must be executed before the slicing criterion in all executions. The CEB slice would contain the whole program.

Therefore, the purpose of our slicing technique is essentially different from previous work: while other approaches try to answer the question "*what parts of the program can influence the value of this variable at this point?*", our technique tries to answer the question "*what parts of the program must be executed before this point? and what parts of the program can be executed before this point?*". Therefore, our slicing criterion is different, but also the data structure we use for slicing is different. In contrast to

previous work, we do not use a PDG like graph, and use instead a CFG like graph, because we focus on control flow rather than control and data dependence.

Despite the problem being undecidable (see Section 1), determining the MEB and CEB slices can be very useful and has many different applications such as debugging, program comprehension, program specialization and program simplification. Surprisingly, to the best of our knowledge, our approach is the first to address the problem in a concurrent and explicitly synchronized context. In fact, the data structure most similar to the CSCFG is the SCFG by Callahan and Sublok [5] (see Section 3 for a detailed description and formalization of this data structure, and a comparison with our CSCFG). Unfortunately, the SCFG does not take the calling context into account and thus it is not appropriate for the MEB and CEB analyses.

Our technique is not the first approach that applies program slicing to CSP specifications. Program slicing has also been applied to CSP by Bruckner and Wehrheim who introduced a method to slice CSP-OZ specifications [3]. Nevertheless, their approach ignores CSP synchronization and focus instead on the OZ's variables. As in previous approaches, their slicing criterion is a LTL formulae constructed with OZ's variables; and they use the standard PDG to compute the slice with a backwards reachability analysis.

7. Conclusions

This work defines two new static analyses that can be applied to languages with explicit synchronization such as CSP. Both techniques are based on program slicing. In particular, we introduce a method to slice CSP specifications, in such a way that, given a CSP specification and a slicing criterion, we produce a slice such that (i) it is a subset of the specification (i.e., it is produced by deleting some parts of the original specification); (ii) it contains all the parts of the specification that must be executed (in any execution) before the slicing criterion (MEB analysis); and (iii) we can also produce an augmented slice that also contains those parts of the specification that could be executed before the slicing criterion (CEB analysis).

We have presented two algorithms to compute the MEB and CEB analyses based on a new data structure, the CSCFG, that has shown to be more precise than the previously used SCFG. The advantage of the CSCFG is that it cares about contexts, and thus it is able to distinguish between different contexts in which a process is called. This new data structure has been formalized in the paper and compared with the predecessor SCFG.

We have built a tool that implements all the data structures and algorithms defined in the paper; and we have integrated it into the system ProB. This tool is called SOC, and it is now distributed as a part of ProB. Finally, a number of experiments conducted with SOC have been presented and discussed. These experiments demonstrated the usefulness of the technique for different applications such as debugging, program comprehension, program specialization and program simplification.

8. Acknowledgments

We want to thank Mark Fontaine for his help in the implementation. He adapted the ProB module that detects source code positions. We also thank the anonymous referees of LOPSTR'08 [19] and PEPM'09 [18] for many useful comments in a preliminary version

of this article.

References

[1] D. Binkley and K. B. Gallagher. Program slicing. *Advances in Computers*, vol. 43, pp. 1–50, 1996.

[2] T. Bolognesi and E. Brinksma. Introduction to the ISO Specification Language LOTOS. In *Computer Networks*, vol. 14, pp. 25–59, 1987.

[3] I. Brückner and H. Wehrheim. Slicing CSP-OZ Specifications for Verification. Technical report, SFB/TR 14 AVACS. Accessible via `http://www.avacs.org/`, 2005.

[4] M. Butler and M. Leuschel. Combining CSP and B for specification and property verification. In *Proceedings of Formal Methods 2005*, LNCS 3582, pp. 221–236, Springer-Verlag, Newcastle, 2005.

[5] D. Callahan and J. Sublok. Static analysis of low-level synchronization. In *Proceedings of the 1988 ACM SIGPLAN and SIGOPS workshop on Parallel and Distributed Debugging (PADD'88)*, pp. 100–111, New York, NY, USA, 1988.

[6] J. Cheng. Slicing concurrent programs - a graph-theoretical approach. In *Automated and Algorithmic Debugging*, pp. 223–240, 1993.

[7] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, vol. 9(3), pp. 319–349, 1987.

[8] D. Goswami and R. Mall. Fast Slicing of Concurrent Programs. In *Proceedings of the 6th International Conference on High Performance Computing*, pp. 38–42, 1999.

[9] M. J. Harrold, G. Rothermel, and S. Sinha. Computation of interprocedural control dependence. In *International Symposium on Software Testing and Analysis*, pp. 11–20, 1998.

[10] C. A. R. Hoare. Communicating sequential processes. *Communications ACM*, vol. 26(1), pp. 100–106, 1983.

[11] G. J. Holzmann. Design and Validation of Computer Protocols. Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[12] K. M. Kavi, F. T. Sheldon, and B. Shirazi. Reliability analysis of CSP specifications using petri nets and markov processes. In *Proceedings 28th Annual Hawaii International Conference on System Sciences. Software Technology*, vol. 2, pp. 516–524, Wailea, HI, 1995.

[13] J. Krinke. Static slicing of threaded programs. In *Workshop on Program Analysis For Software Tools and Engineering*, pp. 35–42, 1998.

[14] J. Krinke. Context-sensitive slicing of concurrent programs. *ACM SIGSOFT Software Engineering Notes*, vol. 28(5), 2003.

[15] P. Ladkin and B. Simons. Static deadlock analysis for csp-type communications. *Responsive Computer Systems (Chapter 5)*, Kluwer Academic Publishers, 1995.

[16] M. Leuschel and M. Butler. ProB: an automated analysis toolset for the B method. *Journal of Software Tools for Technology Transfer*, vol. 10(2), pp. 185–203, 2008.

[17] M. Leuschel and M. Fontaine. Probing the depths of CSP-M: A new FDR-compliant validation tool. In *Proceedings of the 10th International Conference on Formal Engineering Methods*, LNCS 5256, pp. 278–297. Springer-Verlag, 2008.

[18] M. Leuschel, M. Llorens, J. Oliver, J. Silva, and S. Tamarit. SOC: a slicer for CSP specifications. In *Proceedings of the 2009 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM'09)*, pp. 165–168, Savannah, GA, USA, 2009.

[19] M. Leuschel, M. Llorens, J. Oliver, J. Silva, and S. Tamarit. The MEB and CEB Static Analysis for CSP Specifications. In *Post-proceedings of the 18th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'08), Revised and Selected Papers*, LNCS 5438, pp. 103–118, Springer-Verlag, 2009.

[20] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes. In *Journal of Information and Computation*, Academic Press, vol. 100(1), pp. 1–77, 1992.

[21] M. G. Nanda and S. Ramesh. Slicing concurrent programs. In *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 180–190, New York, NY, USA, 2000.

[22] V. Natarajan and G. J. Holzmann. Outline for an Operational Semantics of Promela. In *The SPIN Verification Systems. DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, AMS vol. 32, pp. 133152, 1997.

[23] G. Naumovich and G. S. Avrunin. A conservative data flow algorithm for detecting all pairs of statements that may happen in parallel. *SIGSOFT Software Engineering Notes*, vol. 23(6), pp. 24–34, 1998.

[24] A.W. Roscoe. *The Theory and Practice of Concurrency.* Prentice-Hall, 2005.

[25] A. W. Roscoe, P. H. B. Gardiner, M. Goldsmith, J. R. Hulance, D. M. Jackson, and J. B. Scattergood. Hierarchical compression for model-checking CSP or how to check 10^{20} dining philosophers for deadlock. In *Proceedings of the First International Workshop Tools and Algorithms for Construction and Analysis of Systems*, pp. 133–152, 1995.

[26] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, vol. 3, pp. 121–189, 1995.

[27] M. Weiser. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method.* PhD thesis, The University of Michigan, 1979.

[28] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, vol. 10(4), pp. 352–357, 1984.

[29] J. Zhao, J. Cheng, and K. Ushijima. Slicing concurrent logic programs. In *Proceedings of the 2nd Fuji International Workshop on Functional and Logic Programming*, pp. 143–162, 1997.

[30] J. Zhao. Slicing aspect-oriented software. In *Proceedings of the 10th IEEE International Workshop on Programming Comprehension*, pp. 251–260, 2002.

A. Proofs of Technical Results

In order to prove Lemmas 1-4 and Theorems 8-11, we first introduce and prove some auxiliary lemmas (Lemmas 5-8) that are needed in their proofs.

Lemma 5. *Let $\mathcal{S}$ be a CSP specification and $s \xrightarrow{\Theta} s'$, a rewriting step performed with the instrumented semantics, with $s = (Ctrl_\alpha, \omega)$ and $s' = (Ctrl'_\varphi, \omega')$. Then, $first(\alpha) \in \omega'$.*

Proof. We proceed by analyzing all possible rules applied in the rewriting step. Considering the semantics in Figure 4 the following cases are possible:

- In the cases of (Process Call), (Parameterized Process Call), (Prefixing), (SKIP), (STOP), (Internal Choice 1 and 2), (Conditional Choice 1 and 2), (External Choice 1, 2, 3 and 4), (Synchronized Parallelism 1, 2 and 3), (Hiding 1, 2 and 3) and (Renaming 1, 2 and 3) the lemma is true straightforwardly from the instrumented semantics definition, Definition 2 (rewriting step) and Definition 3 (control flow).
- In the case of (Synchronized Parallelism 4), this implies that in some previous rewriting steps rules (Synchronized Parallelism 1) and (Synchronized Parallelism 2) were applied. Then, the lemma trivially holds.
- (Sequential Composition 1). If we assume that $Ctrl = P;Q$ only contains one single ; operator, then we have a rewriting step of the form:

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

  Thus, the lemma holds by applying any of the previous rules to $(P, \omega) \xrightarrow{a \text{ or } \tau} (P', \omega')$. Contrarily, if $Ctrl$ contains more than one ; operator we know that the number of ; operators is finite because $\mathcal{S}$ is finite. Therefore, we can apply rule (Sequential Composition 1) a finite number of times and then any of the previous rules must be applied thus the lemma will eventually hold.
- (Sequential Composition 2). This rule can only be applied after (Sequential Composition 1). Therefore, $first(\alpha) \in \omega'$ because it was included in a previous rewriting step. Hence, the lemma holds.

□

Lemma 6. *Let $\mathcal{S}$ be a CSP specification, $\mathcal{G} = (N, E_c, E_l, E_s)$ the CSCFG associated with $\mathcal{S}$, and $s_i \longrightarrow s_{i+1}$, $0 \leqslant i < n$, a simple rewriting step of $\mathcal{D} = s_0 \longrightarrow \ldots \longrightarrow s_{n+1}$, $n \geqslant 0$,* a derivation *of $\mathcal{S}$ performed with the instrumented semantics, where $s_i = (Ctrl_\alpha, \omega)$ and $s_{i+1} = (Ctrl'_\varphi, \omega')$. Then, $\exists\ \pi = n_j \mapsto^* n_k \in E_c$, $n_j, n_k \in N$, with $l(n_j) = first(\alpha)$ and $l(n_k) = first(\varphi)$.*

Proof. In one simple rewriting step, only one of the following rules can be applied (note that (SKIP), (STOP) and (Synchronized Parallelism 4) cannot be applied because they would always correspond to the last rewriting step $s_n \longrightarrow s_{n+1}$):

- (Process Call) If $Ctrl_\alpha = M_\alpha$, this rule adds to ω the specification position α of M, and the control changes to $rhs(M)_\varphi$. By Definition 8, the context sensitive control of n_j can pass to n_{j+1} with $l(n_j) = first(\alpha) = \alpha \in \mathcal{C}alls(\mathcal{S})$ and $l(n_{j+1}) =$ "$start\ l(n_j)$", i.e., $\alpha \mapsto$ "$start\ \alpha$"; and the context sensitive control of n_{j+1} can pass to $n_{j+2} = n_k$ with $l(n_{j+2}) = first(\varphi)$, with $\varphi = (lit(\alpha), \Lambda)$, i.e., $first(\alpha) \mapsto$ "$start\ \alpha$" $\mapsto first(\varphi) \in E_c$.

– (Parameterized Process Call) It is completely analogous to (Process Call).

– (Prefixing) If $Ctrl_\alpha = a_\beta \rightarrow_\alpha P_\varphi$, this rule adds to ω the specification positions of the prefix and the prefixing operator, $\alpha = (M, w)$ and $\beta = \mathit{first}(\alpha) = (M, w.1)$ respectively, and the control changes to P_φ, $\varphi = (M, w.2)$. By Definition 7 (using item 2 of Definition 3) and Definition 8, the context sensitive control of n_j can pass to n_{j+1} with $l(n_j) = \beta$ and $l(n_{j+1}) = \alpha$, i.e., $\beta \mapsto \alpha \in E_c$. And by Definition 7 (using item 4 of Definition 3) and Definition 8, the context sensitive control of n_{j+1} can pass to $n_{j+2} = n_k$ with $l(n_{j+1}) = \alpha$ and $l(n_{j+2}) = first(\varphi)$, i.e., $\mathit{first}(\alpha) \mapsto \alpha \mapsto \mathit{first}(\varphi) \in E_c$.

– (Internal Choice 1 and 2) If $Ctrl_\alpha = P \sqcap_\alpha Q$, with this rule the specification position of the choice operator $\alpha = (M, w)$ is added to ω, and one of the two processes P_{φ_1} or Q_{φ_2} is added to the control, with $\varphi_1 = (M, w.1)$ and $\varphi_2 = (M, w.2)$. By Definition 7 (using item 1 of Definition 3) and Definition 8, the context sensitive control of n_j can pass to n_{j+1} and to n_{j+2} with $l(n_j) = \mathit{first}(\alpha) = \alpha$ and $l(n_{j+1}) = first(\varphi_1)$ and $l(n_{j+2}) = \mathit{first}(\varphi_2)$, i.e., $\mathit{first}(\alpha) \mapsto \mathit{first}(\varphi_1) \in E_c$ and $\mathit{first}(\alpha) \mapsto \mathit{first}(\varphi_2) \in E_c$.

– (Conditional Choice 1 and 2) These rules are completely analogous to (Internal Choice 1 and 2).

□

For the following lemma we need to provide a notion of height of a rewriting step. The *height* of a rewriting step $s \xrightarrow{\Theta} t$ is defined as:

$$h(s \xrightarrow{\Theta} t) = \begin{cases} 0 & \text{if } \Theta = \emptyset \\ 1 + max(\{h(s' \xrightarrow{\Theta'} t') \mid s' \xrightarrow{\Theta'} t' \in \Theta\}) & \text{otherwise} \end{cases}$$

Lemma 7. *Let $\mathcal{S}$ be a CSP specification, $\mathcal{G} = (N, E_c, E_l, E_s)$ the CSCFG associated with $\mathcal{S}$, and $s_i \xrightarrow{\Theta_i} s_{i+1}$, $i \geqslant 0$, a rewriting step of $\mathcal{D} = s_0 \longrightarrow \ldots \longrightarrow s_{n+1}$, $n \geqslant 0$,* a derivation *of $\mathcal{S}$ performed with the instrumented semantics, where Θ_i is non empty, $s_i = (Ctrl_\alpha, \omega)$ and $s_{i+1} = (Ctrl'_\varphi, \omega')$. Then, $\forall \gamma \in \omega' \backslash \omega : \exists\, \pi = n_j \mapsto^* n_k \in E_c$, $n_j, n_k \in N$, $k \geqslant 1$, with $l(n_j) = \mathit{first}(\alpha)$ and $l(n_k) = \gamma$.*

Proof. Firstly, we know that if Θ_i is not empty, rules (Process Call), (Parameterized Process Call), (Prefixing), (SKIP), (STOP), (Internal Choice 1 and 2), (Conditional Choice 1 and 2) and (Synchronized Parallelism 4) could not be applied. Then, one of the other rules of the instrumented semantics must be applied.

We prove this lemma by induction on the *height* of the rewriting step. The base case happens when the *height* is one, i.e., the rewriting step is of the form $\dfrac{s_j \overset{a \text{ or } \tau \text{ or } \checkmark}{\longrightarrow} s_{j+1}}{s_i \overset{a \text{ or } \tau \text{ or } \checkmark}{\longrightarrow} s_{i+1}}$.

In one rewriting step of height one, one of the following rules must be applied:

– (External Choice 1, 2, 3 and 4) If $Ctrl_\alpha = P \Box_\alpha Q$, one of these rules can be applied. If event τ happens, rules (Process Call), (STOP), (Internal Choice 1 or 2) or (Conditional Choice 1 or 2) can be applied. If event a happens, rule (Prefixing) is applied. If event $\checkmark$ happens, rules (SKIP) or (Synchronized Parallelism 4) are applied.

  - If (Process Call) is applied, then the rewriting step is:

$$\frac{(P_{\varphi_1}, \omega) \xrightarrow{\tau} (rhs(P)_\beta, \omega \cup \{\varphi_1\})}{(P_{\varphi_1} \Box_\alpha Q_{\varphi_2}, \omega) \xrightarrow{\tau} (rhs(P) \Box_\alpha Q, \omega \cup \{\alpha, \varphi_1\})}$$

By Lemma 6, $\mathit{first}(\varphi_1) \mapsto$ "$\mathit{start}\ \varphi_1$" $\mapsto \mathit{first}(\beta) \in E_c$. By Lemma 5 and by Definition 7 (using item 1 of Definition 3) and Definition 8, the context sensitive control of n_j can pass to n_{j+1} and to n_{j+2} with $l(n_j) = \mathit{first}(\alpha) = \alpha$ and $l(n_{j+1}) = first(\varphi_1) = \varphi_1$ and $l(n_{j+2}) = \mathit{first}(\varphi_2)$, i.e., $\mathit{first}(\alpha) \mapsto^* \mathit{first}(\varphi_1) \in E_c$ and $\mathit{first}(\alpha) \mapsto^* \mathit{first}(\varphi_2) \in E_c$.

- If (STOP) is applied, then the rewriting step is:

$$\frac{(STOP_{\varphi_1}, \omega) \xrightarrow{\tau} (\bot, \omega \cup \{\varphi_1\})}{(STOP_{\varphi_1} \Box_\alpha Q_{\varphi_2}, \omega) \xrightarrow{\tau} (\bot \Box_\alpha Q, \omega \cup \{\alpha, \varphi_1\})}$$

By Lemma 5 and by Definition 7 (using item 1 of Definition 3) and Definition 8, the context sensitive control of n_j can pass to n_{j+1} and to n_{j+2} with $l(n_j) = \mathit{first}(\alpha) = \alpha$ and $l(n_{j+1}) = \mathit{first}(\varphi_1) = \varphi_1$ and $l(n_{j+2}) = \mathit{first}(\varphi_2)$, i.e., $\mathit{first}(\alpha) \mapsto^* \mathit{first}(\varphi_1) \in E_c$ and $\mathit{first}(\alpha) \mapsto^* \mathit{first}(\varphi_2) \in E_c$.

- If (Internal Choice 1 or 2) is applied, then the rewriting step is:

$$\frac{(R_{\beta_1} \sqcap_{\varphi_1} S_{\beta_2}, \omega) \xrightarrow{\tau} (R_{\beta_1}, \omega \cup \{\varphi_1\})}{((R \sqcap_{\varphi_1} S) \Box_\alpha Q_{\varphi_2}, \omega) \xrightarrow{\tau} (R \Box_\alpha Q_{\varphi_2}, \omega \cup \{\alpha, \varphi_1\})}$$

By Lemma 6, $\mathit{first}(\varphi_1) \mapsto \mathit{first}(\beta_1) \in E_c$ and $\mathit{first}(\varphi_1) \mapsto \mathit{first}(\beta_2) \in E_c$. By Lemma 5 and by Definition 7 (using item 1 of Definition 3) and Definition 8, the context sensitive control of n_j can pass to n_{j+1} and to n_{j+2} with $l(n_j) = \mathit{first}(\alpha) = \alpha$ and $l(n_{j+1}) = first(\varphi_1) = \varphi_1$ and $l(n_{j+2}) = \mathit{first}(\varphi_2)$, i.e., $\mathit{first}(\alpha) \mapsto^* \mathit{first}(\varphi_1) \in E_c$ and $\mathit{first}(\alpha) \mapsto^* \mathit{first}(\varphi_2) \in E_c$.

- If (Conditional Choice 1 or 2) is applied, then these rules are completely analogous to (Internal Choice 1 and 2).

- If (Prefixing) is applied, then the rewriting step is:

$$\frac{(a_{\alpha_1} \rightarrow_{\varphi_1} R_{\beta_1}, \omega) \xrightarrow{a} (R_{\beta_1}, \omega \cup \{\alpha_1, \varphi_1\})}{((a_{\alpha_1} \rightarrow_{\varphi_1} R_{\beta_1}) \Box_\alpha Q_{\varphi_2}, \omega) \xrightarrow{a} (R_{\beta_1}, \omega \cup \{\alpha, \alpha_1, \varphi_1\})}.$$

By Lemma 6, $\mathit{first}(\varphi_1) \mapsto \varphi_1 \mapsto \mathit{first}(\beta_1) \in E_c$ where $\mathit{first}(\varphi_1) = \alpha_1$. By Lemma 5 and by Definition 7 (using item 1 of Definition 3) and Definition 8, the context sensitive control of n_j can pass to n_{j+1} and to n_{j+2} with $l(n_j) = \mathit{first}(\alpha) = \alpha$ and $l(n_{j+1}) = first(\varphi_1)$ and $l(n_{j+2}) = \mathit{first}(\varphi_2)$, i.e., $\mathit{first}(\alpha) \mapsto^* \mathit{first}(\varphi_1) \in E_c$ and $\mathit{first}(\alpha) \mapsto^* \mathit{first}(\varphi_2) \in E_c$.

- If (SKIP) is applied, then the rewriting step is:

$$\frac{(SKIP_{\varphi_1}, \omega) \xrightarrow{\checkmark} (\top, \omega \cup \{\varphi_1\})}{(SKIP_{\varphi_1} \Box_\alpha Q_{\varphi_2}, \omega) \xrightarrow{\checkmark} (\top, \omega \cup \{\alpha, \varphi_1\})}$$

By Lemma 5 and by Definition 7 (using item 1 of Definition 3) and Definition 8, the context sensitive control of n_j can pass to n_{j+1} and to n_{j+2} with $l(n_j) = \mathit{first}(\alpha) = \alpha$ and $l(n_{j+1}) = first(\varphi_1) = \varphi_1$ and $l(n_{j+2}) = \mathit{first}(\varphi_2)$, i.e., $\mathit{first}(\alpha) \mapsto^* \mathit{first}(\varphi_1) \in E_c$ and $\mathit{first}(\alpha) \mapsto^* \mathit{first}(\varphi_2) \in E_c$.

- If (Synchronized Parallelism 4) is applied, then the rewriting step is:

$$\frac{(\top||\top,\omega) \xrightarrow{\checkmark} (\top,\omega)}{((\top||\top)\Box_\alpha Q,\omega) \xrightarrow{\checkmark} (\top,\omega \cup \{\alpha\})}$$

If the left(right) process of the choice is $(\top||\top)$, it means that in some previous rewriting steps rules (Synchronized Parallelism 1) and (Synchronized Parallelism 2) were applied. Then, the lemma trivially holds.

– (Synchronized Parallelism 1 and 2) If $Ctrl_\alpha = P \underset{X_\alpha}{\|} Q$, this rule can be applied. If event $a \notin X$ happens, rule (Prefixing) is applied. If event τ happens, rules (Process Call), (SKIP), (STOP), (Internal Choice 1 or 2), (Conditional Choice 1 or 2) or (Synchronized Parallelism 4) can be applied.

- If (Prefixing) is applied, then the rewriting step is:

$$\frac{(a_{\alpha_1} \rightarrow_{\varphi_1} P'_{\beta_1},\omega) \xrightarrow{a} (P',\omega \cup \{\alpha_1,\varphi_1\})}{(a_{\alpha_1} \rightarrow_{\varphi_1} P'_{\beta_1} \underset{X_\alpha}{\|} Q_{\varphi_2},\omega) \xrightarrow{a} (P' \underset{X}{\|} Q,\omega \cup \{\alpha,\alpha_1,\varphi_1\})}\ a \notin X$$

  By Lemma 6, $\mathit{first}(\varphi_1) \mapsto \varphi_1 \mapsto \mathit{first}(\beta_1) \in E_c$ where $\mathit{first}(\varphi_1) = \alpha_1$. By Lemma 5 and by Definition 7 (using item 1 of Definition 3) and Definition 8, the context sensitive control of n_j can pass to n_{j+1} and to n_{j+2} with $l(n_j) = \mathit{first}(\alpha) = \alpha$ and $l(n_{j+1}) = first(\varphi_1)$ and $l(n_{j+2}) = \mathit{first}(\varphi_2)$, i.e., $\mathit{first}(\alpha) \mapsto^* \mathit{first}(\varphi_1) \in E_c$ and $\mathit{first}(\alpha) \mapsto^* \mathit{first}(\varphi_2) \in E_c$.

- If (Process Call) is applied, then the rewriting step is:

$$\frac{(P_{\varphi_1},\omega) \xrightarrow{\tau} (rhs(P)_\beta,\omega \cup \{\varphi_1\})}{(P_{\varphi_1} \underset{X_\alpha}{\|} Q_{\varphi_2},\omega) \xrightarrow{\tau} (rhs(P) \underset{X}{\|} Q,\omega \cup \{\alpha,\varphi_1\})}$$

  By Lemma 6, $\mathit{first}(\varphi_1) \mapsto$ "*start* φ_1" $\mapsto \mathit{first}(\beta) \in E_c$. By Lemma 5 and by Definition 7 (using item 1 of Definition 3) and Definition 8, the context sensitive control of n_j can pass to n_{j+1} and to n_{j+2} with $l(n_j) = \mathit{first}(\alpha) = \alpha$ and $l(n_{j+1}) = first(\varphi_1) = \varphi_1$ and $l(n_{j+2}) = \mathit{first}(\varphi_2)$, i.e., $\mathit{first}(\alpha) \mapsto^* \mathit{first}(\varphi_1) \in E_c$ and $\mathit{first}(\alpha) \mapsto^* \mathit{first}(\varphi_2) \in E_c$.

- If (SKIP) is applied, then the rewriting step is:

$$\frac{(SKIP_{\varphi_1},\omega) \xrightarrow{\checkmark} (\top,\omega \cup \{\varphi_1\})}{(SKIP_{\varphi_1} \underset{X_\alpha}{\|} Q_{\varphi_2},\omega) \xrightarrow{\tau} (\top \underset{X}{\|} Q,\omega \cup \{\alpha,\varphi_1\})}$$

  By Lemma 5 and by Definition 7 (using item 1 of Definition 3) and Definition 8, the context sensitive control of n_j can pass to n_{j+1} and to n_{j+2} with $l(n_j) = \mathit{first}(\alpha) = \alpha$ and $l(n_{j+1}) = first(\varphi_1) = \varphi_1$ and $l(n_{j+2}) = \mathit{first}(\varphi_2)$, i.e., $\mathit{first}(\alpha) \mapsto^* \mathit{first}(\varphi_1) \in E_c$ and $\mathit{first}(\alpha) \mapsto^* \mathit{first}(\varphi_2) \in E_c$.

- If (STOP) is applied, then the rewriting step is:

$$\frac{(STOP_{\varphi_1},\omega) \xrightarrow{\tau} (\bot,\omega \cup \{\varphi_1\})}{(STOP_{\varphi_1} \underset{X_\alpha}{\|} Q_{\varphi_2},\omega) \xrightarrow{\tau} (\bot \underset{X}{\|} Q,\omega \cup \{\alpha,\varphi_1\})}$$

By Lemma 5 and by Definition 7 (using item 1 of Definition 3) and Definition 8, the context sensitive control of n_j can pass to n_{j+1} and to n_{j+2} with $l(n_j) = \mathit{first}(\alpha) = \alpha$ and $l(n_{j+1}) = first(\varphi_1) = \varphi_1$ and $l(n_{j+2}) = \mathit{first}(\varphi_2)$, i.e., $\mathit{first}(\alpha) \mapsto^* \mathit{first}(\varphi_1) \in E_c$ and $\mathit{first}(\alpha) \mapsto^* \mathit{first}(\varphi_2) \in E_c$.

- If (Internal Choice 1 or 2) is applied, then the rewriting step is:

$$\frac{(R_{\beta_1} \sqcap_{\varphi_1} S_{\beta_2}, \omega) \xrightarrow{\tau} (R_{\beta_1}, \omega \cup \{\varphi_1\})}{((R \sqcap_{\varphi_1} S) \underset{X_\alpha}{\parallel} Q_{\varphi_2}, \omega) \xrightarrow{\tau} (R \underset{X_\alpha}{\parallel} Q_{\varphi_2}, \omega \cup \{\alpha, \varphi_1\})}$$

  By Lemma 6, $\mathit{first}(\varphi_1) \mapsto \mathit{first}(\beta_1) \in E_c$ and $\mathit{first}(\varphi_1) \mapsto \mathit{first}(\beta_2) \in E_c$. By Lemma 5 and by Definition 7 (using item 1 of Definition 3) and Definition 8, the context sensitive control of n_j can pass to n_{j+1} and to n_{j+2} with $l(n_j) = \mathit{first}(\alpha) = \alpha$ and $l(n_{j+1}) = first(\varphi_1) = \varphi_1$ and $l(n_{j+2}) = \mathit{first}(\varphi_2)$, i.e., $\mathit{first}(\alpha) \mapsto^* \mathit{first}(\varphi_1) \in E_c$ and $\mathit{first}(\alpha) \mapsto^* \mathit{first}(\varphi_2) \in E_c$.

- If (Conditional Choice 1 or 2) is applied, then these rules are completely analogous to (Internal Choice 1 and 2).

- If (Synchronized Parallelism 4) is applied, then the rewriting step is:

$$\frac{(\top || \top, \omega) \xrightarrow{\checkmark} (\top, \omega)}{((\top || \top) \underset{X_\alpha}{\parallel} Q, \omega) \xrightarrow{\tau} (\top \underset{X}{\parallel} Q, \omega \cup \{\alpha\})}$$

  If the left(right) process of the parallelism is $(\top || \top)$, it means that in some previous rewriting steps rules (SP1) and (SP2) were applied. Then, the lemma trivially holds.

– (Synchronized Parallelism 3) If $Ctrl_\alpha = P \underset{X_\alpha}{\parallel} Q$, this rule can be applied. When event $a \in X$ happens, only rule (Prefixing) can be applied. Then the rewriting step is:

$$\frac{(a_{\alpha_1} \rightarrow_{\varphi_1} P'_{\beta_1}, \omega) \xrightarrow{a} (P'_{\beta_1}, \omega \cup \{\alpha_1, \varphi_1\}) \qquad (a_{\alpha_2} \rightarrow_{\varphi_2} Q'_{\beta_2}, \omega) \xrightarrow{a} (Q'_{\beta_2}, \omega \cup \{\alpha_2, \varphi_2\})}{((a_{\alpha_1} \rightarrow_{\varphi_1} P'_{\beta_1}) \underset{X_\alpha}{\parallel} (a_{\alpha_2} \rightarrow_{\varphi_2} Q'_{\beta_2}), \omega) \xrightarrow{a} (P' \underset{X}{\parallel} Q', \omega \cup \{\alpha, \alpha_1, \varphi_1, \alpha_2, \varphi_2\})}$$

  By Lemma 6, $\mathit{first}(\varphi_1) \mapsto \varphi_1 \mapsto \mathit{first}(\beta_1) \in E_c$ where $\mathit{first}(\varphi_1) = \alpha_1$ and $\mathit{first}(\varphi_2) \mapsto \varphi_2 \mapsto \mathit{first}(\beta_2) \in E_c$ where $\mathit{first}(\varphi_2) = \alpha_2$. By Lemma 5 and by Definition 7 (using item 1 of Definition 3) and Definition 8, the context sensitive control of n_j can pass to n_{j+1} and to n_{j+2} with $l(n_j) = \mathit{first}(\alpha) = \alpha$ and $l(n_{j+1}) = first(\varphi_1)$ and $l(n_{j+2}) = \mathit{first}(\varphi_2)$, i.e., $\mathit{first}(\alpha) \mapsto^* \mathit{first}(\varphi_1) \in E_c$ and $\mathit{first}(\alpha) \mapsto^* \mathit{first}(\varphi_2) \in E_c$.

– (Sequential Composition 1) If $Ctrl_\alpha = P ;_\alpha Q$, this rule can be applied. When event a happens, only rule (Prefixing) can be applied. If event τ happens, rules (Process Call), (STOP), (Internal Choice 1 or 2) or (Conditional Choice 1 or 2) can be applied.

  - If (Prefixing) is applied, then the rewriting step is:

$$\frac{(a_{\alpha_1} \rightarrow_{\varphi_1} P'_{\beta_1}, \omega) \xrightarrow{a} (P'_{\beta_1}, \omega \cup \{\alpha_1, \varphi_1\})}{((a_{\alpha_1} \rightarrow_{\varphi_1} P'_{\beta_1}) ;_\alpha Q_{\varphi_2}, \omega) \xrightarrow{a} (P' ; Q, \omega \cup \{\alpha_1, \varphi_1\})}$$

By Lemma 6, $\mathit{first}(\varphi_1) \mapsto \varphi_1 \mapsto \mathit{first}(\beta_1) \in E_c$ where $\mathit{first}(\varphi_1) = \alpha_1$. By Definition 7 (using item 3 of Definition 3) and Definition 8, the context sensitive control of n_j can pass to n_{j+1} with $l(n_j) = \mathit{last}(\varphi_1) = \mathit{last}(\beta_1)$ and $l(n_{j+1}) = \alpha$, i.e., $\mathit{last}(\beta_1) \mapsto \alpha \in E_c$.

- If (Process Call) is applied, then the rewriting step is:

$$\frac{(P_{\varphi_1}, \omega) \xrightarrow{\tau} (rhs(P)_{\beta_1}, \omega \cup \{\varphi_1\})}{(P_{\varphi_1};_\alpha Q_{\varphi_2}, \omega) \xrightarrow{\tau} (rhs(P); Q, \omega \cup \{\varphi_1\})}$$

By Lemma 6, $\mathit{first}(\varphi_1) \mapsto$ "$\mathit{start}\ \varphi_1$" $\mapsto \mathit{first}(\beta_1) \in E_c$. By Definition 7 (using item 3 of Definition 3) and Definition 8, the context sensitive control of n_j can pass to n_{j+1} with $l(n_j) = \mathit{last}(\varphi_1) = \mathit{last}(\beta_1)$ and $l(n_{j+1}) = \alpha$, i.e., $\mathit{last}(\beta_1) \mapsto \alpha \in E_c$.

- If (STOP) is applied, then the rewriting step is:

$$\frac{(STOP_{\varphi_1}, \omega) \xrightarrow{\tau} (\perp, \omega \cup \{\varphi_1\})}{(STOP_{\varphi_1};_\alpha Q_{\varphi_2}, \omega) \xrightarrow{\tau} (\perp; Q, \omega \cup \{\varphi_1\})}$$

By Definition 7, Definition 8 and Lemma 5, the context sensitive control of n_j can pass to n_{j+1} with $l(n_j) = first(\alpha)$ and $l(n_{j+1}) = \varphi_1$, i.e., $first(\alpha) \mapsto \varphi_1 \in E_c$.

- If (Internal Choice 1 or 2) is applied, then the rewriting step is:

$$\frac{(R_{\beta_1} \sqcap_{\varphi_1} S_{\beta_2}, \omega) \xrightarrow{\tau} (R_{\beta_1}, \omega \cup \{\varphi_1\})}{((R_{\beta_1} \sqcap_{\varphi_1} S_{\beta_2});_\alpha Q_{\varphi_2}, \omega) \xrightarrow{\tau} (R; Q, \omega \cup \{\varphi_1\})}$$

By Lemma 6, $\mathit{first}(\varphi_1) \mapsto \mathit{first}(\beta_1) \in E_c$ and $\mathit{first}(\varphi_1) \mapsto \mathit{first}(\beta_2) \in E_c$. By Definition 7 (using item 3 of Definition 3) and Definition 8, the context sensitive control of n_j can pass to n_{j+1} and to n_{j+2} with $l(n_j) = \mathit{last}(\varphi_1) = \mathit{last}(\beta_1)$, $l(n_{j+1}) = \alpha$ and $l(n_{j+2}) = \mathit{last}(\varphi_2) = \mathit{last}(\beta_2)$, i.e., $\mathit{last}(\beta_1) \mapsto \alpha \in E_c$ and $\mathit{last}(\beta_1) \mapsto \alpha \in E_c$.

- If (Conditional Choice 1 or 2) is applied, then these rules are completely analogous to (Internal Choice 1 and 2).

– (Sequential Composition 2) If $Ctrl_\alpha = P;_\alpha Q$ and this rule can be applied, P will be $SKIP$ or $\top||\top$, i.e., rules (SKIP) or (Synchronized Parallelism 4) can be applied.

- If (SKIP) is applied, then the rewriting step is:

$$\frac{(SKIP_{\varphi_1}, \omega) \xrightarrow{\checkmark} (\top, \omega \cup \{\varphi_1\})}{(SKIP_{\varphi_1};_\alpha Q_{\varphi_2}, \omega) \xrightarrow{\tau} (Q, \omega \cup \{\alpha, \varphi_1\})}$$

By Definition 7 (using item 4 of Definition 3) and Definition 8, the context sensitive control of n_j can pass to n_{j+1} with $l(n_j) = \alpha$ and $l(n_{j+1}) = \mathit{first}(\varphi_2)$, i.e., $\alpha \mapsto \mathit{first}(\varphi_2) \in E_c$.

- If (Synchronized Parallelism 4) is applied, then the rewriting step is:

$$\frac{(\top||\top, \omega) \xrightarrow{\checkmark} (\top, \omega)}{((\top||\top);_\alpha Q_{\varphi_2}, \omega) \xrightarrow{\tau} (Q, \omega \cup \{\alpha\})}$$

By Definition 7 (using item 4 of Definition 3) and Definition 8, the context sensitive control of of n_j can pass to n_{j+1} with $l(n_j) = \alpha$ and $l(n_{j+1}) = \mathit{first}(\varphi_2)$, i.e., $\alpha \mapsto \mathit{first}(\varphi_2) \in E_c$.

- (Hiding 1) If $Ctrl_\alpha = P\backslash_\alpha B$, this rule can be applied. When event $a \in B$ happens, only rule (Prefixing) can be applied. The rewriting step is:

$$\frac{(a_{\alpha_1} \rightarrow_{\varphi_1} P'_{\beta_1}, \omega) \xrightarrow{a} (P'_{\beta_1}, \omega \cup \{\alpha_1, \varphi_1\})}{((a_{\alpha_1} \rightarrow_{\varphi_1} P'_{\beta_1})\backslash_\alpha B, \omega) \xrightarrow{\tau} (P'\backslash_\alpha B, \omega \cup \{\alpha, \alpha_1, \varphi_1\})}$$

By Lemma 6, $\mathit{first}(\varphi_1) \mapsto \varphi_1 \mapsto \mathit{first}(\beta_1) \in E_c$ where $\mathit{first}(\varphi_1) = \alpha_1$. By Definition 7 (using item 5 of Definition 3) and Definition 8, the context sensitive control of n_j can pass to n_{j+1} with $l(n_j) = first(\alpha) = \alpha$ and $l(n_{j+1}) = first(\varphi_1)$, i.e., $\alpha \mapsto^* \mathit{first}(\varphi_1) \in E_c$.

- (Hiding 2) If $Ctrl_\alpha = P\backslash_\alpha B$, this rule can be applied. When event $a \notin B$ happens, only rule (Prefixing) can be applied. If event τ happens, rules (Process Call), (STOP), (Internal Choice 1 or 2) or (Conditional Choice 1 or 2) can be applied.

  - If (Prefixing) is applied, then the rewriting step is:

$$\frac{(a_{\alpha_1} \rightarrow_{\varphi_1} P'_{\beta_1}, \omega) \xrightarrow{a} (P'_{\beta_1}, \omega \cup \{\alpha_1, \varphi_1\})}{((a_{\alpha_1} \rightarrow_{\varphi_1} P'_{\beta_1})\backslash_\alpha B, \omega) \xrightarrow{a} (P'\backslash_\alpha B, \omega \cup \{\alpha, \alpha_1, \varphi_1\})} \; a \notin B$$

    By Lemma 6, $\mathit{first}(\varphi_1) \mapsto \varphi_1 \mapsto \mathit{first}(\beta_1) \in E_c$ where $\mathit{first}(\varphi_1) = \alpha_1$. By Definition 7 (using item 5 of Definition 3) and Definition 8, the context sensitive control of n_j can pass to n_{j+1} with $l(n_j) = first(\alpha) = \alpha$ and $l(n_{j+1}) = first(\varphi_1)$, i.e., $\alpha \mapsto^* \mathit{first}(\varphi_1) \in E_c$.

  - If (STOP) is applied, then the rewriting step is:

$$\frac{(STOP_{\varphi_1}, \omega) \xrightarrow{\tau} (\perp, \omega \cup \{\varphi_1\})}{(STOP_{\varphi_1}\backslash_\alpha B, \omega) \xrightarrow{\tau} (\perp\backslash_\alpha B, \omega \cup \{\alpha, \varphi_1\})}$$

    By Definition 7 (using item 5 of Definition 3) and Definition 8, the context sensitive control of n_j can pass to n_{j+1} with $l(n_j) = \alpha$ and $l(n_{j+1}) = first(\varphi_1) = \varphi_1$, i.e., $\alpha \mapsto^* \mathit{first}(\varphi_1) \in E_c$.

  - If (Internal Choice 1 or 2) is applied, then the rewriting step is:

$$\frac{(R_{\beta_1} \sqcap_{\varphi_1} S_{\beta_2}, \omega) \xrightarrow{\tau} (R_{\beta_1}, \omega \cup \{\varphi_1\})}{((R_{\beta_1} \sqcap_{\varphi_1} S_{\beta_2})\backslash_\alpha B, \omega) \xrightarrow{\tau} (R_{\beta_1}\backslash_\alpha B, \omega \cup \{\alpha, \varphi_1\})}$$

    By Lemma 6, $\mathit{first}(\varphi_1) \mapsto \mathit{first}(\beta_1) \in E_c$ and $\mathit{first}(\varphi_1) \mapsto \mathit{first}(\beta_2) \in E_c$. By Definition 7 (using item 5 of Definition 3) and Definition 8, the context sensitive control of n_j can pass to n_{j+1} with $l(n_j) = first(\alpha) = \alpha$ and $l(n_{j+1}) = first(\varphi_1) = \varphi_1$, i.e., $\alpha \mapsto^* \mathit{first}(\varphi_1) \in E_c$.

  - If (Conditional Choice 1 or 2) is applied, then these rules are completely analogous to (Internal Choice 1 and 2).

– (Hiding 3) If $Ctrl_\alpha = P\backslash_\alpha B$ and this rule can be applied, P will be $SKIP$ or $\top||\top$, i.e., rules (SKIP) or (Synchronized Parallelism 4) can be applied.

- If (SKIP) is applied, then the rewriting step is:

$$\frac{(SKIP_{\varphi_1}, \omega) \xrightarrow{\checkmark} (\top, \omega \cup \{\varphi_1\})}{(SKIP_{\varphi_1}\backslash_\alpha B, \omega) \xrightarrow{\checkmark} (\top, \omega \cup \{\alpha, \varphi_1\})}$$

  By Definition 7 (using item 5 of Definition 3) and Definition 8, the context sensitive control of n_j can pass to n_{j+1} with $l(n_j) = first(\alpha) = \alpha$ and $l(n_{j+1}) = first(\varphi_1) = \varphi_1$, i.e., $\alpha \mapsto^* \mathit{first}(\varphi_1) \in E_c$.

- If (Synchronized Parallelism 4) is applied, then the rewriting step is:

$$\frac{(\top\|_{\varphi_1}\top, \omega) \xrightarrow{\checkmark} (\top, \omega)}{((\top\|_{\varphi_1}\top)\backslash_\alpha B, \omega) \xrightarrow{\checkmark} (\top, \omega \cup \{\alpha\})}$$

  If the process $(\top||\top)$ is in the control, it means that in some previous rewriting step rules (Synchronized Parallelism 1), (Synchronized Parallelism 2) and/or (Synchronized Parallelism 3) were applied and $\varphi_1 \in \omega$. By Definition 7 (using item 5 of Definition 3) and Definition 8, the context sensitive control of n_j can pass to n_{j+1} with $l(n_j) = first(\alpha) = \alpha$ and $l(n_{j+1}) = first(\varphi_1) = \varphi_1$, i.e., $\alpha \mapsto^* \mathit{first}(\varphi_1) \in E_c$.

– (Renaming 1, 2 and 3) These rules are completely analogous to (Hiding 1, 2 and 3).

We assume as the induction hypothesis that the lemma holds in rewriting steps of height n, and we prove that it also holds in a rewriting step of height $n + 1$, i.e., the rewriting step is of the form:

$$\frac{\dfrac{\Theta}{s_j \overset{a \text{ or } \tau \text{ or } \checkmark}{\longrightarrow} s_{j+1}}}{s_i \overset{a \text{ or } \tau \text{ or } \checkmark}{\longrightarrow} s_{i+1}}$$

where Θ is of height $n-1$ ($n \geqslant 2$), $s_i = (Ctrl_\alpha, \omega)$, $s_j = (Ctrl_\beta, \omega)$, $s_{j+1} = (Ctrl'_\delta, \omega')$ and $s_{i+1} = (Ctrl''_\varphi, \omega'')$.

In order to prove this lemma, we take advantage of the induction hypothesis that ensures that $\forall \gamma \in \omega'\backslash\omega :\ \exists\ \pi = n_j \mapsto^* n_k \in E_c$, $n_j, n_k \in N$, $k \geqslant 1$, with $l(n_j) = \mathit{first}(\beta)$ and $l(n_k) = \gamma$. Therefore, we need to prove that:

1. $\exists\ \pi_1 = n_i \mapsto^* n_j \in E_c$, $n_i \in N$, with $l(n_i) = \mathit{first}(\alpha)$, and
2. $\forall \gamma \in \omega''\backslash\omega' :\ \exists\ \pi_2 = n_i \mapsto^* n_k \in E_c$.

In order to prove, item 1, one of the following rules can be applied:

– (External Choice 1, 2, 3 and 4) Trivially, using item 1 of Definition 3, $\mathit{first}(\alpha) \mapsto^* \mathit{first}(\beta) \in E_c$.

– (Synchronized Parallelism 1, 2 and 3) These rules are completely analogous to (External Choice 1, 2, 3 and 4).

– (Hiding 1 and 2) Trivially, using item 5 of Definition 3, $\mathit{first}(\alpha) \mapsto^* \mathit{first}(\beta) \in E_c$.

– (Renaming 1 and 2) These rules are completely analogous to (Hiding 1 and 2).

– (Sequential Composition 1 and 2) In these cases $first(\alpha) = first(\beta)$ and the lemma trivially holds.

In order to prove, item 2, we consider the following two cases:

– One of these rules is applied: (External Choice 1, 2, 3 and 4), (Synchronized Parallelism 1, 2 and 3), (Hiding 1 and 2) and (Renaming 1 and 2). In these cases, $\omega''\backslash\omega' = \{\alpha\}$ and $\alpha = first(\alpha)$. Therefore, the lemma holds.

– (Sequential Composition 1) This case is trivial because $\omega''\backslash\omega' = \varnothing$.

□

Lemma 8. *Let $\mathcal{S}$ be a CSP specification, $\mathcal{G} = (N, E_c, E_l, E_s)$ the CSCFG associated with $\mathcal{S}$, and $s_i \xrightarrow{\Theta_i} s_{i+1}$, $i > 0$, a rewriting step of $\mathcal{D} = s_0 \longrightarrow \ldots \longrightarrow s_{n+1}$, $n \geqslant 0$, a* derivation *of $\mathcal{S}$ performed with the instrumented semantics, where $s_i = (Ctrl_\alpha, \omega_i)$ and $s_{i+1} = (Ctrl'_\varphi, \omega_{i+1})$. Then, $\exists\ \pi = n_j \mapsto^* n_k \in E_c$, $n_j, n_k \in N$, with $l(n_j) \in \omega_i$ and $l(n_k) = first(\alpha)$.*

Proof. Let us consider the rewriting step $s_{i-1} = (Ctrl_\zeta, \omega_{i-1}) \xrightarrow{\Theta_{i-1}} s_i = (Ctrl_\alpha, \omega_i)$.

If $\Theta_{i-1} = \varnothing$, rules (Process Call), (Parameterized Process Call), (Prefixing), (SKIP), (STOP), (Internal Choice 1 and 2), (Conditional Choice 1 and 2) and (Synchronized Parallelism 4) can be applied. In these cases, by Lemma 5 and Lemma 6, the lemma trivially holds.

If $\Theta_{i-1} \neq \varnothing$ and one of the rules (External Choice 1, 2, 3 and 4), (Synchronized Parallelism 1, 2, 3 and 4), (Sequential Composition 1), (Hiding 1, 2 and 3) or (Renaming 1, 2 and 3) is applied, we know by Lemma 5 and by Lemma 7 that $first(\zeta) \in \omega_i$ and that $\exists\ \pi = n_j \mapsto^* n_k \in E_c$, $n_j, n_k \in N$, $k \geqslant 1$, with $l(n_j) \in \omega_i$ and $l(n_k) = first(\alpha)$.

If rule (Sequential Composition 2) is applied, we know by Lemma 5 that $first(\zeta) \in \omega_i$ and by Definition 7 (using item 4 of Definition 3) and Definition 8, the context sensitive control of n_j can pass to n_k with $l(n_j) = \zeta \in \omega_i$, $lit(l(n_j)) =$; and $l(n_k) = first(\alpha)$, i.e., $\zeta \mapsto first(\alpha) \in E_c$.

□

The following lemma ensures that the CSCFG is complete: all possible derivations of a CSP specification $\mathcal{S}$ are represented in the CSCFG associated to $\mathcal{S}$.

Lemma 1. *Let $\mathcal{S}$ be a CSP specification, $\mathcal{D} = s_0 \longrightarrow \ldots \longrightarrow s_{n+1}$, $n \geqslant 0$, a* derivation *of $\mathcal{S}$ performed with the instrumented semantics, where $s_0 = (rhs(\texttt{MAIN})_\alpha, \varnothing)$ and $s_{n+1} = (P_\varphi, \omega)$, and $\mathcal{G} = (N, E_c, E_l, E_s)$ the CSCFG associated with $\mathcal{S}$. Then, $\forall \gamma \in \omega :\ \exists\ \pi = n_1 \mapsto^* n_k \in E_c$, $n_1, n_k \in N$, $k \geqslant 1$, with $l(n_1) = first(\alpha)$ and $l(n_k) = \gamma$.*

Proof. We prove this lemma by induction on the length of the derivation $\mathcal{D}$. The base case happens when the length of $\mathcal{D}$ is one. The initial state is $s_0 = (rhs(\texttt{MAIN})_\alpha, \varnothing)$. The final state is $s_1 = (P_\varphi, \omega)$. In one rewriting step, one of the following rules must be applied:

- (Process Call) If $rhs(\texttt{MAIN}) = Q_\alpha$, this rule adds to ω the specification position α of P, and the control changes to $rhs(Q)_\varphi$, i.e., $s_1 = (rhs(Q)_\varphi, \{\alpha\})$. By Definition 8, the context sensitive control of n_0 can pass to n_1 with $l(n_0)$ = "*start* $(\texttt{MAIN}, 0)$" and $l(n_1) = first((\texttt{MAIN}, \Lambda)) = \alpha$, i.e., "*start* $(\texttt{MAIN}, 0)$" $\mapsto \alpha \in E_c$. By Definition 3, $first((\texttt{MAIN}, \Lambda)) = \alpha = first(\alpha)$; by Definition 8, the context sensitive control of n_1 can pass to n_2 with $l(n_2)$ = "*start* $(\texttt{Q}, 0)$" and by Lemma 6, $\exists\ \pi = n_1 \mapsto n_2 \mapsto n_3 \in E_c$ with $l(n_1) = first(\alpha)$ and $l(n_k) = first(\varphi)$, i.e., $\alpha \mapsto$ "*start* $(\texttt{Q}, 0)$" $\mapsto first(\varphi) \in E_c$.
- (Parameterized Process Call) It is completely analogous to (Process Call).
- (Prefixing) If $rhs(\texttt{MAIN}) = a_\beta \to_\alpha P_\varphi$, this rule adds to ω the specification positions of the prefix and the prefixing operator, α and β respectively, and the control changes to P_φ, i.e., $s_1 = (P_\varphi, \{\alpha, \beta\})$. By Definition 8 and Definition 3, the context sensitive control of n_0 can pass to n_1 with $l(n_0)$ = "*start* $(\texttt{MAIN}, 0)$" and $l(n_1) = first((\texttt{MAIN}, \Lambda)) = (\texttt{MAIN}, 1) = \beta$, i.e., "*start* $(\texttt{MAIN}, 0)$" $\mapsto \beta \in E_c$. By Definition 7 (using item 2 of Definition 3) and Definition 8, the context sensitive control of n_1 can pass to n_2 with $l(n_1) = \beta$ and $l(n_2) = \alpha$, i.e., $\beta \mapsto \alpha \in E_c$. By Lemma 6, $\exists\ \pi = n_1 \mapsto n_2 \mapsto n_3 \in E_c$ with $l(n_1) = first(\alpha) = \beta$ and $l(n_3) = first((\texttt{MAIN}, 2)) = first(\varphi)$, i.e., $\beta \mapsto \alpha \mapsto first(\varphi) \in E_c$.
- (SKIP) If $rhs(\texttt{MAIN}) = \texttt{SKIP}_\alpha$, applying this rule the specification position α of SKIP is added to ω, the control changes to $\top$, i.e., $s_1 = (\top, \{\alpha\})$, and the derivation finishes. By Definition 8, the context sensitive control of n_0 can pass to n_1 with $l(n_0)$ = "*start* $(\texttt{MAIN}, 0)$" and $l(n_1) = first((\texttt{MAIN}, \Lambda)) = \alpha$, i.e., "*start* $(\texttt{MAIN}, 0)$" $\mapsto \alpha \in E_c$. And by Definition 7 and Definition 8, the context sensitive control of n_1 can pass to n_2 with $l(n_2)$ = "*end* $(\texttt{MAIN}, 0)$", i.e., $\alpha \mapsto$ "*end* $(\texttt{MAIN}, 0)$" $\in E_c$.
- (STOP) If $rhs(\texttt{MAIN}) = \texttt{STOP}_\alpha$, applying this rule the specification position α of STOP is added to ω, the control changes to $\bot$, i.e., $s_1 = (\bot, \{\alpha\})$, and the derivation finishes. By Definition 8, the context sensitive control of n_0 can pass to n_1 with $l(n_0)$ = "*start* $(\texttt{MAIN}, 0)$" and $l(n_1) = first((\texttt{MAIN}, \Lambda)) = \alpha$, i.e., "*start* $(\texttt{MAIN}, 0)$" $\mapsto \alpha \in E_c$.
- (Internal Choice 1 and 2) If $rhs(\texttt{MAIN}) = P_{\varphi_1} \sqcap_\alpha Q\varphi_2$, with this rule the specification position of the choice operator α is added to ω, and one of the two processes P or Q is added to the control, i.e., $s_1 = (P_{\varphi_1}, \{\alpha\})$ or $s_1 = (Q_{\varphi_2}, \{\alpha\})$. By Definition 8, the context sensitive control of n_0 can pass to n_1 with $l(n_0)$ = "*start* $(\texttt{MAIN}, 0)$" and $l(n_1) = first((\texttt{MAIN}, \Lambda)) = \alpha$, i.e., "*start* $(\texttt{MAIN}, 0)$" $\mapsto \alpha \in E_c$. By Lemma 6, $\exists\ \pi = n_1 \mapsto n_2 \in E_c$ with $l(n_2) = \varphi = first((\texttt{MAIN}, 1)) = first(\varphi_1)$ and $\exists\ \pi = n_1 \mapsto n_3 \in E_c$ with $l(n_3) = first((\texttt{MAIN}, 2)) = first(\varphi_2)$, i.e., $\alpha \mapsto first(\varphi_1) \in E_c$ and $\alpha \mapsto first(\varphi_2) \in E_c$.
- (Conditional Choice 1 and 2) These rules are completely analogous to (Internal Choice 1 and 2).
- (External Choice 1, 2, 3 and 4) If $rhs(\texttt{MAIN}) = P \,\square_\alpha Q$, with one of these rules the specification position of the choice operator α and the set of executed specification positions of process P or Q is added to ω. By Definition 8, the context sensitive control of n_0 can pass to n_1 with $l(n_0)$ = "*start* $(\texttt{MAIN}, 0)$" and

$l(n_1) = first((\texttt{MAIN}, \Lambda)) = \alpha$, i.e., "$start$ $(\texttt{MAIN}, 0)$" $\mapsto \alpha \in E_c$. By Lemma 7, $\forall \gamma \in \omega : \exists\, \pi = n_1 \mapsto^* n_k \in E_c$ with $l(n_k) = \gamma$.

- (Synchronized Parallelism 1 and 2) If $rhs(\texttt{MAIN}) = P \underset{X_\alpha}{\|} Q$, with one of these rules the specification position of the parallelism operator together with the specification positions executed of the corresponding process are added to ω. By Definition 8, the context sensitive control of n_0 can pass to n_1 with $l(n_0) =$ "$start$ $(\texttt{MAIN}, 0)$" and $l(n_1) = first((\texttt{MAIN}, \Lambda)) = \alpha$, i.e., "$start$ $(\texttt{MAIN}, 0)$" $\mapsto \alpha \in E_c$. By Lemma 7, $\forall \gamma \in \omega : \exists\, \pi = n_1 \mapsto^* n_k \in E_c$ with $l(n_k) = \gamma$.
- (Synchronized Parallelism 3) If $rhs(\texttt{MAIN}) = P \underset{X_\alpha}{\|} Q$, with this rule the specification position of the parallelism operator together with the specification positions executed of the two processes are added to ω. By Definition 8, the context sensitive control of n_0 can pass to n_1 with $l(n_0) =$ "$start$ $(\texttt{MAIN}, 0)$" and $l(n_1) = first((\texttt{MAIN}, \Lambda)) = \alpha$, i.e., "$start$ $(\texttt{MAIN}, 0)$" $\mapsto \alpha \in E_c$. By Lemma 7, $\forall \gamma \in \omega : \exists\, \pi = n_1 \mapsto^* n_k \in E_c$ with $l(n_k) = \gamma$.
- (Synchronized Parallelism 4) This rule does not add specification positions to ω.
- (Sequential Composition 1) If $rhs(\texttt{MAIN}) = P\ ;_\alpha Q$, this rule can be applied. The control changes to $P'\ ;_\alpha Q$ and the executed specification positions of P will be added to ω. By Definition 8, the context sensitive control of n_0 can pass to n_1 with $l(n_0) =$ "$start$ $(\texttt{MAIN}, 0)$" and $l(n_1) = first((\texttt{MAIN}, \Lambda)) = first((\texttt{MAIN}, 1))$, i.e., "$start$ $(\texttt{MAIN}, 0)$" $\mapsto first((\texttt{MAIN}, 1)) \in E_c$. By Lemma 7, $\forall \gamma \in \omega : \exists\, \pi = n_1 \mapsto^* n_k \in E_c$ with $l(n_k) = \gamma$.
- (Sequential Composition 2) For this rule to be applied, $rhs(\texttt{MAIN}) = SKIP_\beta\ ;_\alpha Q_\varphi$. The control changes to Q and $\omega = \{\alpha, \beta\}$, i.e., $s_1 = (Q_\varphi, \{\alpha, \beta\})$. By Definition 8, the context sensitive control of n_0 can pass to n_1 with $l(n_0) =$ "$start$ $(\texttt{MAIN}, 0)$" and $l(n_1) = first((\texttt{MAIN}, \Lambda)) = first((\texttt{MAIN}, 1)) = \beta$, i.e., "$start$ $(\texttt{MAIN}, 0)$" $\mapsto \beta \in E_c$. By Lemma 7, $n_1 \mapsto n_2 \in E_c$ with $l(n_2) = \alpha$. And by Definition 7 (using item 4 of Definition 3) and Definition 8, the context sensitive control of n_2 can pass to n_3 with $l(n_2) = \alpha$ and $l(n_3) = \varphi = first((\texttt{MAIN}, 2))$, i.e., $\alpha \mapsto \varphi \in E_c$.
- (Hiding 1 and 2) If $rhs(\texttt{MAIN}) = P \backslash_\alpha B$ and one of these rules is applied, ω is increased with the specification position α of the hiding operator and with the specification positions of the developed process P. By Definition 8, the context sensitive control of n_0 can pass to n_1 with $l(n_0) =$ "$start$ $(\texttt{MAIN}, 0)$" and $l(n_1) = first((\texttt{MAIN}, \Lambda)) = \alpha$, i.e., "$start$ $(\texttt{MAIN}, 0)$" $\mapsto \alpha \in E_c$. By Lemma 7, $\forall \gamma \in \omega : \exists\, \pi = n_1 \mapsto^* n_k \in E_c$ with $l(n_k) = \gamma$.
- (Hiding 3) For this rule to be applied, $rhs(\texttt{MAIN}) = SKIP_\beta \backslash_\alpha B$. The control changes to $\top$, $\omega = \{\alpha, \beta\}$, i.e., $s_1 = (\top, \{\alpha, \beta\})$, and the derivation finishes. By Definition 8, the context sensitive control of n_0 can pass to n_1 with $l(n_0) =$ "$start$ $(\texttt{MAIN}, 0)$" and $l(n_1) = first((\texttt{MAIN}, \Lambda)) = \alpha$, i.e., "$start$ $(\texttt{MAIN}, 0)$" $\mapsto \alpha \in E_c$. By Lemma 7, $n_1 \mapsto n_2 \in E_c$ with $l(n_2) = first(\beta) = \beta$. And by Definition 7 (using item 6 of Definition 3) and Definition 8, the context sensitive control of n_2 can pass to n_3 with $l(n_3) =$ "end $(\texttt{MAIN}, \Lambda)$", and the context sensitive control of n_3 can pass to n_4 with $l(n_4) =$ "end $(\texttt{MAIN}, 0)$", i.e., $\beta \mapsto$ "end $(\texttt{MAIN}, 0)$" $\mapsto$ "end $(\texttt{MAIN}, 0)$" $\in E_c$,

– (Renaming 1 and 2) These rules are completely analogous to (Hiding 1 and 2).

– (Renaming 3) It is completely analogous to the rule (Hiding 3).

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 $i+1$. Let us consider $s_i = (Ctrl_\beta, \omega) \xrightarrow{\Theta_i} s_{i+1} = (Ctrl'_\delta, \omega')$. By the induction hypothesis we know that there exists a path from $first((\texttt{MAIN}, \Lambda))$ to all positions in ω. By Lemma 8, we know that there exists a path from a specification position in ω to $first(\beta)$. And by Lemma 7, we know that there exists a path from $first(\beta)$ to all positions in $\omega \backslash \omega'$. Therefore, the lemma holds. □

Lemma 3. *Let $\mathcal{S}$ be a CSP specification and let $\mathcal{G} = (N, E_c, E_l, E_s)$ be the CSCFG associated with $\mathcal{S}$ according to Definition 8. If $n \mapsto n' \in E_c$ then n must be executed before n' in all executions.*

Proof. We prove this lemma by induction on the length of a path $\pi_{\mathcal{S}tart} = n_1 \mapsto^* n_m$ in the CSCFG $\mathcal{G}$. Firstly, because all nodes in $\mathcal{S}tart(\mathcal{S})$ are not executable, we remove them from the path. Hence, we assume that for all $1 \leqslant j < m$ we have $l(n_j) \notin \mathcal{S}tart(\mathcal{S})$. We refer to this reduced path as π.

The base case happens when the length of π is one. The first node of the graph is always "$start$ $(\texttt{MAIN}, 0)$", hence, $\pi = n \mapsto n'$. Therefore, by Definition 8, $l(n) = first(\texttt{MAIN}, \Lambda)$.

In this situation, $lit(l(n))$ can be:

- If $lit(l(n)) = a$, with $a \in \Sigma$, then by Definition 3 (item 2), we have that $lit(l(n')) = \rightarrow$. In the semantics, the only rule applicable is Prefixing. Therefore, n must be executed before n'.
- If $lit(l(n)) = STOP$, by Definition 3 and Definition 8 there is no control from $STOP$. Therefore, $n' \notin N$ and thus this case is not possible.
- If $lit(l(n)) \in \{\sqcap, \square, \lessdot\gtrdot, |||, ||\}$ then it is possible to apply Choice or Parallelism.
  - If we have a choice, by Definition 3 (item 1), $l(n') \in \{first((\texttt{MAIN}, 1)), first((\texttt{MAIN}, 2))\}$. In the semantics, the only rule applicable is a choice. Therefore, n must be executed before n'.
  - Analogously, if we have a parallelism, by Definition 3 (item 1), we have that $l(n') \in \{first((\texttt{MAIN}, 1)), first((\texttt{MAIN}, 2))\}$. In the semantics, the only rule applicable is a synchronized parallelism. Therefore, n must be executed before n'.
- If $lit(l(n)) \in \{\backslash, [\![ ]\!]\}$, then by Definition 3 (item 5), we have that $lit(l(n')) = first((\texttt{MAIN}, 1))$. In the semantics, the only rule applicable is Hiding or Renaming. Therefore, n must be executed before n'.
- If $lit(l(n)) = SKIP$ then two cases are possible:
  - by Definition 3 (item 3), $lit(l(n')) = ;$ because $last(l(n)) = l(n)$. In the semantics, the only rule applicable is Sequential Composition 2 and this necessarily implies that $SKIP$ is executed before. Thus, n must be executed before n'.

- by Definition 8, we have in $\pi_{\mathcal{S}tart}$ that $lit(l(n')) =$ "*end* $(\texttt{MAIN}, 0)$" and in π, n' does not exist. In fact, in the semantics, the only rule applicable is SKIP that finishes the execution.

- If $lit(l(n)) = P$ with $P \in \mathcal{P}$ then, $l(n) = \mathit{first}(\texttt{MAIN}, \Lambda) = (\texttt{MAIN}, \Lambda)$, and by Definition 8, we have $\pi_{\mathcal{S}tart} =$ "*start* $(\texttt{MAIN}, 0)$" $\mapsto (\texttt{MAIN}, \Lambda) \mapsto$ "*start* $(\texttt{MAIN}, \Lambda)$" $\mapsto \mathit{first}((P, \Lambda))$. Therefore, $\pi = (\texttt{MAIN}, \Lambda) \mapsto \mathit{first}((P, \Lambda))$. In the semantics, the only rule applicable is Process Call that changes the root position in the control to (P, Λ). Then, by Lemma 5 we know that the next rewriting step will have $\mathit{first}((P, \Lambda)) \in \omega$.

We assume as the induction hypothesis that the lemma holds for a path π with length k. We prove that it also holds for a path with length $k + 1$. Therefore, it is enough to prove that n_k must be executed before n_{k+1} in all executions with $n_k \mapsto n_{k+1} \in E_c$.

We analyze each possible case with respect to $lit(l(n_k))$. All cases are analogous to the base case except three:

- If $lit(l(n_k)) = \rightarrow$, with $l(n_k) = (M, w)$, then by Definition 3 (item 4), we have that $l(n_{k+1}) = \mathit{first}((M, w.2))$. In the semantics, the last rule applied was Prefixing, because it is the only rule that introduces $\rightarrow$. This rule puts in the control $(M, w.2)$. Therefore, by Lemma 5 we know that the next rewriting step will have $\mathit{first}((M, w.2)) \in \omega$. Therefore, n_k must be executed before n_{k+1}.

- If $lit(l(n_k)) =$; then it is completely analogous to the previous case but now the last rule applied is Sequential Composition 2.

- If $lit(l(n_k)) = SKIP$ then two cases are possible:

  - by Definition 3 (item 3), $lit(l(n_{k+1})) =$; because $last(l(n_k)) = l(n_k)$. In the semantics, the only rule applicable is Sequential Composition 2 and this necessarily implies that SKIP is executed before. Thus, n_k must be executed before n_{k+1}.
  - by Definition 8, we have in $\pi_{\mathcal{S}tart}$ that $lit(l(m + 1)) =$ "*end* $(\texttt{MAIN}, 0)$" and in π, $m + 1$ does not exist. In fact, in the semantics, the only rule applicable is SKIP that finishes the execution.

Therefore, the lemma is true. □

Lemma 2. *Let $\mathcal{S}$ be a CSP specification. Then, the execution of Algorithm 1 with $\mathcal{S}$ produces a graph $\mathcal{G}$ that is the CSCFG associated with $\mathcal{S}$ according to Definition 8.*

Proof. Let $\mathcal{G} = (N, E_c, E_l, E_s)$ the CSCFG associated with $\mathcal{S}$ according to Definition 8. And let $\mathcal{G}' = (N', E'_c, E'_l, E'_s)$ the output of Algorithm 1 with input $\mathcal{S}$. We now prove that $\mathcal{G} = \mathcal{G}'$.

In order to prove that both graphs are equivalent, we proceed by case analysis of function *buildGraph*. This is enough to prove the equivalence because this function is used to build the graph associated to the right hand side of all functions. Therefore, we have to prove not only that the graphs are equivalent for each case, but also that the returned values n_{first} and *Last* are the expected ones so that recursion of this function also works and thus the produced graphs are correctly joined to form the final CSCFG.

In all cases E_s will be equivalent to E'_s if the rest of the components of the graph are equivalent, because both are built using the same technique. We assume by induction hypothesis that, in all cases, the values returned by recursive calls are the expected ones. Let us study each case separately:

- Prefixing: In this case $P = X_\alpha \rightarrow_\beta Q$ and $X \in \{a, a?v, a!v\}$. We let $\beta = (M, w)$ thus $\alpha = (M, w.1)$ and the label of Q is $(M, w.2)$. Function *first* returns for this expression $(M, w.1)$. In the function it is represented by the fact that $n_{first} = n_\alpha$ (note that $l(n_\alpha) = \alpha = (M, w.1)$). Function *last* will return the set $last(n2)$ where $l(n2) = (M, w.2)$. In the algorithm, the variable *Last* is bound to $Last_1$, that is a set whose labels correspond to $last(n2)$. The nodes introduced by the algorithm are n_α and n_β. Their labels belong to $\mathcal{P}os(\mathcal{S})$, thus they are in N. The nodes in the set N_1 are also nodes from N. Hence, we can state that $N = N'$. This case introduces two control arcs plus the arcs introduced by the graph of Q. These two control arcs correspond to the second and fourth item of definition 3. The first is represented by $n_\alpha \mapsto n_\beta$, and the second by $n_\beta \mapsto n_{first1}$. Then, we have that $E_c = E'_c$. Finally, E'_l is equal to E_{l1}. Note that prefixing does not introduce loop arcs, so in this case we also have that $E_l = E'_l$.

- Choice and Parallelism: In this case $P = Q\ X_\alpha\ R$ and $X \in \{\sqcap, \Box, ⦓⦔, |||, ||\}$. Let $\alpha = (M, w)$. Hence the labels of Q and R are $(M, w.1)$ and $(M, w.2)$ respectively. Function *first* returns (M, w) for this expression. In the function it is represented by the fact that $n_{first} = n_\alpha$. Function *last* will return the set $last(n1) \cup last(n2)$, where $l(n1) = (M, w.1)$ and $l(n2) = (M, w.2)$, if none of these sets is empty or the operator is not a parallel operator neither an interleaving. Otherwise *last* will return $\varnothing$. In the algorithm, the variable *Last* is bounded to $Last_1 \cup Last_2$ (note that their labels will be all in last(n1) and last(n2)) or $\varnothing$ depending on the same condition. The nodes introduced by the algorithm are n_α and the nodes of sets N_1 and N_2. All these labels belong to $\mathcal{P}os(\mathcal{S})$, so we can state that $N = N'$. This case introduces two control arcs plus the arcs introduced by the graph of Q and R. These two control arcs correspond to the first item of definition 3. They are represented by $n_\alpha \mapsto first_1$ and $n_\alpha \mapsto first_2$. Then, we have that $E_c = E'_c$. Finally, E'_l is equal to $E_{l1} \cup E_{l2}$, hence $E_l = E'_l$.

- Sequential Composition: In this case $P = Q\ ;_\alpha\ R$. Let $\alpha = (M, w)$. Then the labels of Q and R are $(M, w.1)$ and $(M, w.2)$ respectively. Function *first* returns for this expression $first((M, w.1))$. In the function it is represented by the fact that $n_{first} = n_{first1}$. Function *last* will return the set $last(n2)$, where $l(n2) = (M, w.2)$. In the algorithm, it is represented by the fact that the variable *Last* is bounded to $Last_2$ which their node labels are all in $last(n2)$. The nodes introduced by the algorithm are n_α and the sets of nodes N_1 and N_2. All these labels belong to $\mathcal{P}os(\mathcal{S})$, so, we can state that $N = N'$. This case introduces many control arcs plus the arcs introduced by the graph of Q and R. Concretely, it introduces the arc $n_\alpha \mapsto n_{first2}$ and the set E_{c3} where all the nodes belonging to $Last_1$ are joined to node n_α. The former corresponds to the fourth item of definition 3. The latter corresponds to the third item. Then, we have that $E_c = E'_c$. Finally, E'_l is equal to $E_{l1} \cup E_{l2}$, so $E_l = E'_l$.

- Hiding and Renaming: In this case $P = Q\ X_\alpha$ and $X \in \{\backslash, [\![]\!]\}$. Let $\alpha = (M, w)$ then the label of Q is $(M, w.1)$. Function *first* returns for this expression (M, w). In the function, it is represented by the fact that $n_{first} = n_\alpha$ (note that $l(n_\alpha) = \alpha = (M, w)$). Function *last* will return the set $last(n1)$, where $l(n1) = (M, w.1)$. In the algorithm, the variable $Last$ is bounded to $Last_1$ and all their node labels are in $last(n1)$. The nodes introduced by the algorithm are n_α, n_{end} and the nodes of sets N_1. All these labels belong to $\mathcal{P}os(\mathcal{S})$, except n_{end} that belongs to $\mathcal{S}tart(\mathcal{S})$. Thus, we can state that $N = N'$. This case introduces the control arc $n_\alpha \mapsto n_first1$, the arcs of E_{c2} plus the arcs introduced by the graph of Q. The single arc corresponds to the fifth item of definition 3, and the edges in E_{c2} correspond to the sixth item of the same definition. Then, we have that $E_c = E'_c$. Finally, E'_l is equal to $E_{l1} \cup E_{l2}$, so $E_l = E'_l$.
- SKIP and STOP: In this case $P = Q\ X_\alpha$ and $X \in \{SKIP, STOP\}$. Let $\alpha = (M, w)$. Function *first* returns for this expression (M, w). In the function it is represented by the fact that $n_{first} = n_\alpha$ (note that $l(n_\alpha) = \alpha = (M, w)$). Function *last* will return the set $\{(M, w)\}$ if $lit(l(n)) = SKIP$ or $\emptyset$ if $lit(l(n)) = STOP$. In the algorithm, the variable $Last$ is bounded to $\{n_\alpha\}$ or $\emptyset$ with the same conditions. The only node introduced by the algorithm is n_α that belongs to $\mathcal{P}os(\mathcal{S})$. Thus, we can state that $N = N'$. This case does not introduce any control arc. Then, we have that $E_c = E'_c$, because with only one node it is not possible to have control flow. Finally, E'_l is equal to $\emptyset$, so, as it happens in the previous case, $E_l = E'_l$.
- (Parameterized) Process Call: In this case $P = X_\alpha$. Let $\alpha = (M, w)$. Function *first* returns (M, w) for this expression. In the function it is represented by the fact that $n_{first} = n_\alpha$ (note that $l(n_\alpha) = \alpha = (M, w)$). Function *last* will return $\emptyset$ or $\{$"$end(M, w)$"$\}$ depending on whether a node with label "$start(M, w)$" is in $\mathcal{C}on(n)$ or not respectively. This is the same condition that we find in the conditional clause of the algorithm, thus we have a total correspondence. The graph components also depend in this condition, so we are going to distinguish between two cases. First, when the start node is in the context, and second when it is not.
  - The only node introduced by the algorithm is n_α that belongs to $\mathcal{P}os(\mathcal{S})$. Thus, we can state that $N = N'$. This case does not introduce any control arc. Then, we have that $E_c = E'_c$. Finally, E'_l is equal to a set with a unique loop arc from n_α to n_{ctx} (which is the node that makes the condition hold). This arc is the same as the one introduced in the same conditions of definition 8, so $E_l = E'_l$.
  - The nodes introduced by the algorithm are n_α, that belongs to $\mathcal{P}os(\mathcal{S})$, and n_{start} and n_{end} that belongs to $\mathcal{S}tart(\mathcal{S})$. These nodes plus set N_1 form N', so we can state that $N = N'$. The set E'_c if formed by the union of set E_{c1} from the graph of the right-hand side of process X, and set E_{c2}. The latter represents the special control flow stated in the second item of definition 8. Then, we have that $E_c = E'_c$. Finally, E'_l is equal to E_{l1}, so $E_l = E'_l$.

□

Lemma 4. (Finiteness) *Given a specification $\mathcal{S}$, its associated CSCFG is finite.*

Proof. We show first that there does not exist infinite unfolding in a CSCFG. Firstly, the same start process node only appears twice in the same control loop-free path if it belongs to a process which is called from different process calls (i.e., with different specification positions) as it is ensured by Definition 8. Therefore, the number of repeated nodes in the same control loop-free path is limited by the number of different process calls appearing in the program. Moreover, the number of terms in the specification is finite and thus there is a finite number of different process calls. In addition, every process call has only one outgoing arc as it is ensured by the third property of Definition 8. Therefore, the number of paths is finite and the size of every path of the CSCFG is limited.

□

Theorem 8. (Termination of MEB) *The MEB analysis performed by Algorithm 2 terminates.*

Proof. First, we know that N is finite, and thus *blseeds* is also finite because $blseeds \subset N$. Therefore, the first loop (4) always terminates because it is repeated while new nodes are added to *blacklist*; and the number of possible insertions is finite because N is finite. We can ensure that the second loop also terminates due to the invariant $pending \cap Meb = \varnothing$ which is always true at the end of the loop (9). Then, because *Meb* increases in every iteration (7) and the size of N is finite, *pending* will eventually become empty and the loop will terminate.

□

Theorem 10. (Termination of CEB) *The CEB analysis performed by Algorithm 3 terminates.*

Proof. Firstly, the algorithm starts with a call to the function *buildMeb*. By Lemma 8, this call always terminates. Then, the only loops that could cause non-termination are the loop containing sentences (4) to (10) and the loop containing sentences (13) to (18). The first loop is repeated until no new nodes are added to the sets *loopnodes* or *candidates*. We know that *loopnodes* never decreases in the loop; moreover, sentence (4) ensures that $m' \notin loopnodes$, therefore, the number of nodes added to *loopnodes* is finite because the number of nodes in N is finite. Similarly, *candidates* only have a finite number of insertions, and once a node is added to *candidates* it can be removed, but never inserted again because *loopnodes* never decreases. The second loop is analogous to the first one. Therefore, we can ensure that it always terminates by showing that *Ceb* is increased in every iteration with nodes of *pending* that leave *pending* when they are inserted into *Ceb*, see (14) and (17). And, moreover, *pending* can only be increased a limited number of times because it is always increased with nodes which are the successor of a node in *pending* following control arcs. Therefore, because the CSCFG is a tree if we only consider control arcs and N is finite, the size of every branch is finite, and thus, the loop always terminates.

□

Theorem 9. (Completeness of MEB) *Let $\mathcal{S}$ be a specification, $\mathcal{C}$ a slicing criterion for $\mathcal{S}$, and let $\mathcal{MEB}$ be the MEB slice of $\mathcal{S}$ with respect to $\mathcal{C}$. Then, $\mathcal{MEB} \subseteq MEB(\mathcal{S},\mathcal{C})$.*

Proof. (sketch) First, we prove that $MEB(\mathcal{S},\mathcal{C})$ considers all possible executions of the slicing criterion as $\mathcal{MEB}$ does. This depends on function *nodes*, that considers only the first occurrences (starting from `MAIN` and proceeding forwards) of the slicing criterion. Then, it is equivalent to consider the first time that the slicing criterion belongs to ω, which is the stopping condition in the $\mathcal{MEB}$ construction process. Second, as the slicing criterion could happen in different executions, we have to construct a relation between them in order to build the final result. In the case of $\mathcal{MEB}$ the intersection of ω for each execution is considered. However, $MEB(\mathcal{S},\mathcal{C})$ considers the intersection of the specification positions of each node belonging to *buildMeb*(n) where n is a slicing criterion's node. So we have to prove that the result of *buildMeb*(n) will return all (and maybe more) the nodes (and consequently specification positions) that have been executed before it. Function *buildMeb* in step (2) selects all the nodes from `MAIN` to the given node n and to those nodes which are synchronized with it. The rest of the nodes that will be appended in the rest of steps depends mainly on sets *pending* and *sync*. These sets will be formed by those nodes that are synchronized with nodes in set *Meb* or loops which contain at least one node synchronized with a node in *Meb*. Then all possible executed nodes are belonging to *Meb* at the end. The only problem that could arise happens when some nodes are not considered and they are included in the *blacklist*, the set of discarded nodes. However, all the nodes in this set are correctly discarded, because it adds first the given node n and the branches of choices or interleaving which do not reach a node in *Meb*; then, it discards iteratively the nodes under these ones and all that are synchronized with them only if all the other nodes synchronized are also discarded. Therefore, we can conclude that the result will be a superset of $\mathcal{MEB}$. □

Theorem 11. (Completeness of CEB) *Let $\mathcal{S}$ be a specification, $\mathcal{C}$ a slicing criterion for $\mathcal{S}$, and let $\mathcal{CEB}$ be the CEB slice of $\mathcal{S}$ with respect to $\mathcal{C}$. Then, $\mathcal{CEB} \subseteq CEB(\mathcal{S},\mathcal{C})$.*

Proof. (sketch) The first part of the proof is completely analogous to the previous one. The rest of the proof concerns function *buildCeb*. In its first step, a call to function *buildMeb* is made. Consequently all those parts that must be executed before node n will form the initial set for *Ceb*. Then, we have to prove that the rest of steps collects those nodes that could also be executed before the given node n. This group only depends on loops that finish in a node that is in *Ceb*. Then, in step (2) the set *loopnodes* is initialized with the children of the choices in *Ceb* that are not in *Ceb*. After this, an iterative process proceeds forward adding nodes if they could be executed, i.e. if it has not synchronization arcs; or in case it has, their synchronized nodes are in *Ceb* or in *candidates* (a set of nodes waiting for acceptance). In this way, it is assured that only those nodes that could be executed before n are added to *Ceb* in step (11). Finally, the same checking idea is applied to the nodes that are under nodes in *Ceb* (but n), adding iteratively more nodes if the conditions are fulfilled. With this last step, the rest of specification positions that could be executed (those belonging to other threads of execution) is safely added to the set *Ceb*. Then, we can conclude that $CEB(\mathcal{S},\mathcal{C})$ is a superset of $\mathcal{CEB}$. □