The MEB and CEB Static Analysis for CSP Specifications

Michael Leuschel[1], Marisa Llorens[2], Javier Oliver[2],
Josep Silva[2], and Salvador Tamarit[2]

[1] Institut für Informatik, Heinrich-Heine-Universität Düsseldorf, Universitätsstrae 1, D-40225 Düsseldorf , Germany.
leuschel@cs.uni-duesseldorf.de
[2] Technical University of Valencia, Camino de Vera S/N, E-46022 Valencia, Spain.
{mllorens,fjoliver,jsilva,stamarit}@dsic.upv.es

Abstract. This work presents a static analysis technique based on program slicing for CSP specifications. Given a particular event in a CSP specification, our technique allows us to know what parts of the specification must necessarily be executed before this event, and what parts of the specification could be executed before it in some execution. Our technique is based on a new data structure which 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, makes the slicing process more precise.

Keywords: Concurrent Programming, CSP, Program Slicing.

1 Introduction

The *Communicating Sequential Processes* (CSP) [6] language allows us to specify complex systems with multiple interacting processes. The study and transformation of such systems often implies different analyses (e.g., deadlock analysis [10], reliability analysis [7], refinement checking [15], etc.).

In this work we introduce a static analysis technique for CSP which is based on a well-known program comprehension technique called *program slicing* [17].

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 which are (potentially) related with 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) [4] 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), which 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 slicing can be found, e.g., in [16].

Our technique allows us to extract the part of a CSP specification which is related to a given event (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 CSP specification.

In particular, given an event in a specification, our technique allows us to extract those parts of the specification which must be executed before the specified event (thus they are an implicit precondition); and those parts of the specification which could, and could not, be executed before it.

Example 1. Consider the following specification[3] with 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 which finish without any fail.

```
MAIN = (STUDENT || PARENT) || COLLEGE
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 → STOP □ pass → C1
C1 = fail → STOP □ pass → C2
C2 = fail → STOP □ pass → prize → STOP
```

In this specification, 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` (thus the slicing criterion is (`YEAR2`, `fail`)). Our slicing technique automatically extracts the slice composed by the black parts. We can additionally be interested in knowing which 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. Note that, in both cases, the slice produced could be made executable by replacing the removed parts by "`STOP`" or by "→ `STOP`" if the removed expression has a prefix.

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

```
MAIN = P ⊓ Q
```

[3] We refer those readers non familiarized with CSP syntax to Section 2 where we provide a brief introduction to CSP.

```
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 that X is an infinite process which is known to be an undecidable problem [17].

The main contributions of this work are the following:

- We define two new static analyses for CSP and propose algorithms for their implementation. Despite their clear usefulness we have not found these 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 taking into account the context of process calls; and it is particularly interesting for slicing languages with explicit synchronization.
- We have implemented our technique in a prototype integrated in ProB [11, 1, 12]. Preliminary results are very encouraging.

The rest of the paper is organized as follows. In Section 2 we overview the CSP language and introduce some notation. In Section 3 we show that previous data structures used in program slicing are inaccurate 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. Next, we discuss some related works in Section 5. In Section 6 we present our implementation, and, finally, Section 7 concludes.

2 Communicating Sequential Processes

In order to keep the paper self-contained, in the following we recall the syntax of our CSP specifications. We also introduce here some notation that will be used along the paper.

Figure 1 summarizes the syntax constructions used in our CSP specifications. A specification is a finite collection of definitions. The left-hand side of each definition is the name of a different process, which is defined in the right-hand side by means of an expression that can be a call to another process or a combination of the following operators:

Prefixing. It specifies that event a must happen before expression π.
Internal choice. The system chooses (e.g., nondeterministically) to execute one of the two expressions.
External choice. It is identic to internal choice but the choice comes from outside the system (e.g., the user).
Interleaving. Both expressions are executed in parallel and independently.

$S ::= D_1 \dots D_m$	(entire specification)	*Domains*
$D ::= P = \pi$	(definition of a process)	$P, Q, R \dots$ (processes)
$\pi ::= Q$	(process call)	$a, b, c \dots$ (events)
$\mid\ a \rightarrow \pi$	(prefixing)	
$\mid\ \pi_1 \sqcap \pi_2$	(internal choice)	
$\mid\ \pi_1 \Box \pi_2$	(external choice)	
$\mid\ \pi_1 \mid\mid\mid \pi_2$	(interleaving)	
$\mid\ \pi_1 \mid\mid_{\{\overline{a_n}\}} \pi_2$	(synchronized parallelism)	where $\overline{a_n} = a_1, \dots, a_n$
$\mid\ \pi_1 \ ; \ \pi_2$	(sequential composition)	
$\mid\ SKIP$	(skip)	
$\mid\ STOP$	(stop)	

Fig. 1. Syntax of CSP specifications

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

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

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

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

Note, to simplify the presentation we do not yet treat process parameters, nor input and output of data values on channels.

We define the following notation for a given CSP specification $\mathcal{S}$: $\mathcal{P}roc(\mathcal{S})$ and $\mathcal{E}vent(\mathcal{S})$ are, respectively, the sets of all (possible repeated) process calls and events appearing in $\mathcal{S}$. Similarly, $\mathcal{P}roc(P)$ and $\mathcal{E}vent(P)$ with $P \in \mathcal{S}$, are, respectively, the sets of all (possible repeated) processes and events in P. In addition, we define $choices(A)(parallel(A))$, where A is a set of processes, events and operators; as the subset of operators that are either an internal choice or an external choice (an interleaving or a synchronized parallelism).

We use $a_1 \rightarrow^* a_n$ to denote a feasible (sub)execution which leads from a_1 to a_n; and we say that $b \in (a_1 \rightarrow^* a_n)$ iff $b = a_i, 1 \leq i \leq n$. In the following, unless we state the contrary, we will assume that programs start the computation from a distinguished process `MAIN`.

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

Definition 1 (position, specification position).
Positions are represented by a sequence of natural numbers, where Λ denotes the

empty sequence (i.e., the root position). They are used to address subexpressions of an expression viewed as a tree:

$\pi|_\Lambda = \pi$ *for all process* π
$(\pi_1\ op\ \pi_2)|_{1.w} = \pi_1|_w$ *for all operator* $op \in \{\rightarrow, \sqcap, \Box, |||, ||, ;\}$
$(\pi_1\ op\ \pi_2)|_{2.w} = \pi_2|_w$ *for all operator* $op \in \{\rightarrow, \sqcap, \Box, |||, ||, ;\}$

Given a specification $\mathcal{S}$*, we let* $\mathcal{P}os(\mathcal{S})$ *denote the set of all specification positions in* $\mathcal{S}$*. A specification position is a pair* $(P, w) \in \mathcal{P}os(\mathcal{S})$ *that addresses the subexpression* $\pi|_w$ *in the right-hand side of the definition,* $P = \pi$*, of process* P *in* $\mathcal{S}$*.*

Example 2. In the following specification each term has been labelled (in grey color) with its associated specification position so that all labels are unique.

$\texttt{MAIN} =$
$(\texttt{P}_{(Main,1.1)} ||_{\{b\}(Main,1)} \texttt{Q}_{(Main,1.2)})\ ;_{(Main,\Lambda)}\ (\texttt{P}_{(Main,2.1)} ||_{\{a\}(Main,2)} \texttt{R}_{(Main,2.2)})$

$\texttt{P} = \texttt{a}_{(P,1)} \rightarrow_{(P,\Lambda)} \texttt{b}_{(P,2.1)} \rightarrow_{(P,2)} \texttt{SKIP}_{(P,2.2)}$

$\texttt{Q} = \texttt{b}_{(Q,1)} \rightarrow_{(Q,\Lambda)} \texttt{c}_{(Q,2.1)} \rightarrow_{(Q,2)} \texttt{SKIP}_{(Q,2.2)}$

$\texttt{R} = \texttt{d}_{(R,1)} \rightarrow_{(R,\Lambda)} \texttt{a}_{(R,2.1)} \rightarrow_{(R,2)} \texttt{SKIP}_{(R,2.2)}$

We often use indistinguishably an expression and its specification position (e.g., (Q, c) and $(Q, 2.1)$) when it is clear from the context.

3 Context-Sensitive Synchronized Control Flow Graphs

As it is usual in static analysis, we need a data structure able to finitely represent the (often infinite) computations of our specifications. Unfortunately, we cannot use the standard *Control Flow Graph* (CFG) [16], neither the *Interprocedural Control Flow Graph* (ICFG) [5] 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) [8, 9] can represent multiple threads through the use of the so called "*start thread*" and "*end thread*" nodes; but it does not handle synchronizations between threads. Callahan and Sublok introduced a data structure [2], the *Synchronized Control Flow Graph* (SCFG), which explicitly represents synchronizations between threads with a special edge for synchronization flows.

For convenience, the following definition adapts the original definition of SCFG for CSP; and, at the same time, it slightly extends it with the "*start thread*" and "*end thread*" notation from tCFGs.

Definition 2. *(Synchronized Control Flow Graph) Given a CSP specification* $\mathcal{S}$*, its* Synchronized Control Flow Graph *is a directed graph,* $SCFG = (N, E_c, E_s)$ *where nodes* $N = \mathcal{P}os(\mathcal{S}) \cup \mathcal{S}tart(\mathcal{S})$*; and* $\mathcal{S}tart(\mathcal{S}) = \{$"*start* P", "*end* P"$|P \in$

$\mathcal{P}roc(\mathcal{S})\}$. *Edges are divided into two groups,* control-flow edges *(E_c) and* synchronization edges *(E_s). E_s is a set of two-way edges (denoted with $\leftrightarrow$) representing the possible synchronization of two (event) nodes. E_c is a set of one-way edges (denoted with $\mapsto$) such that, given two nodes $n, n' \in N$, $n \mapsto n' \in E_c$ iff one of the following is true:*

- $n = P \wedge n' =$ "*start* P" with $P \in \mathcal{P}roc(\mathcal{S})$
- $n =$ "*start* P" $\wedge\ n' = first((P, \Lambda))$ with $P \in \mathcal{P}roc(\mathcal{S})$
- $n \in \{\sqcap, \Box, |||, ||\} \wedge n' \in \{first(n.1), first(n.2)\}$
- $n \in \{\rightarrow, ;\} \wedge n' = first(n.2)$
- $n = n'.1 \wedge n' = \rightarrow$
- $n \in last(n'.1) \wedge n' =\ ;$
- $n \in last((P, \Lambda)) \wedge n' =$ "*end* P" with $P \in \mathcal{P}roc(\mathcal{S})$

where

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

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

The SCFG can be used for slicing CSP specifications as it is described in the following example.

Example 3. Consider the specification of Example 2 and its associated SCFG shown in Fig. 2 (left); for the sake of clarity we show the term represented by each specification position. If we select the node labelled `c` and traverse the SCFG backwards in order to identify the nodes on which `c` depends, we get the whole graph except nodes `end MAIN`, `end R` and `SKIP` at process `R`.

This example is twofold: on the one hand, it shows that the SCFG could be used for static slicing of CSP specifications. On the other hand, it shows that it is still too imprecise as 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 possible different synchronizations, and, thus, slicing lacks precision. For instance, in Example 2 process `P` is called twice in different contexts. It is first executed in parallel with `Q` producing the synchronization of their `b` events. Then, it is executed in parallel with `R` producing the synchronization of their `a` events. This makes the complete process `R` be part of the slice, which is suboptimal because process `R` is always executed after `Q`.

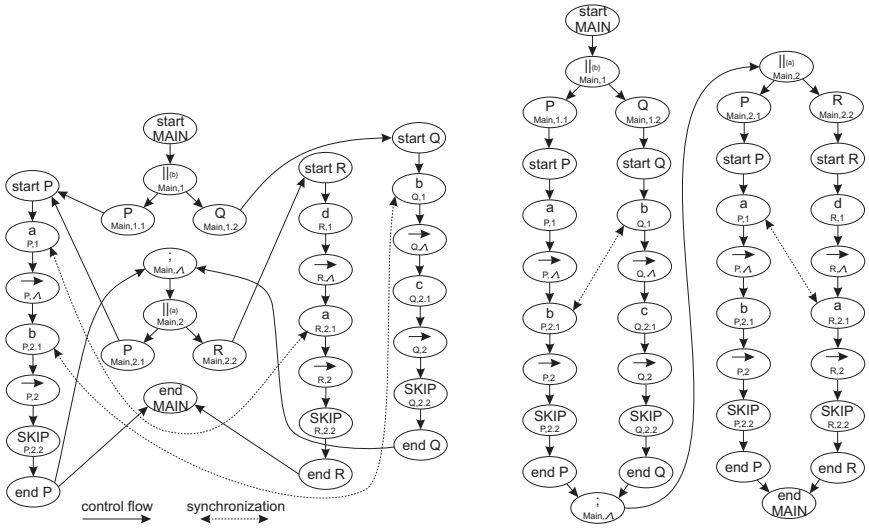

Fig. 2. SCFG (left) and CSCFG (right) of the program in Example 2

To the best of our knowledge, there do not exist other graph representations which face this problem. 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.

Contrarily to the SCFG, inside a CSCFG the same specification position can appear multiple times. Hence, we now use labelled graphs, with nodes labelled by specification positions. Therefore, we use $l(n)$ to refer to the label of node n. We also need to define the context of a node in the graph.

Definition 3. *(Context) A* path *between two nodes n_1 and m is a sequence $l(n_1), \ldots, l(n_k)$ such that $n_k \mapsto m$ and for all $0 < i < k$ we have $n_i \mapsto n_{i+1}$. The path is* loop-free *if for all $i \neq j$ we have $n_i \neq n_j$.*
Given a labelled graph $G = (N, E_c)$ and a node $n \in N$, the context of n, $\mathcal{C}on(n) = \{m \mid l(m) = \text{“}start\ P\text{”}, P \in \mathcal{P}roc(\mathcal{S})$ and exists a loop-free path $\pi = m \mapsto^* n$ with “end P” $\notin \pi\}$.

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 $n \in \mathcal{P}roc(\mathcal{S})$ we can use the context to identify loops.

Definition 4. *(Context-Sensitive Synchronized Control Flow Graph) Given a CSP specification $\mathcal{S}$, a* Context-Sensitive Synchronized Control Flow Graph *$CSCFG = (N, E_c, E_l, E_s)$ is a SCFG graph, except in two aspects:*

1. *There is a special set of* loop edges *(E_l) denoted with $\rightsquigarrow$. $(n_1 \rightsquigarrow n_2) \in E_l$ iff $l(n_1) = P \in \mathcal{P}roc(\mathcal{S})$, $l(n_2) =$ “start P” and $n_2 \in \mathcal{C}on(n_1)$, and*

2. *Two nodes can have the same label. Every node in $\mathcal{S}tart(\mathcal{S})$ has one and only one incoming edge in E_c. Every process call node has one and only one outgoing edge which belongs to either E_c or E_l.*

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, in all executions, if n is executed there must be some n'' which is executed at the same time than n with $n \leftrightarrow n'' \in E_s$.

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 which is executed in a different context is unfolded, thus, slicing does not mix computations. Moreover, it allows to deal with recursion and, at the same time, it prevents from infinite unfolding of process calls; because loop edges prevent from infinite unfolding. One important characteristic of the CSCFG is that loops are unfolded once, and thus 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.

The CSCFG provides a different representation for each context in which a procedure call is made. This can be seen in Fig. 2 (right) where process `P` 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 labelled with the same specification position) and they belong to the same loop. This property ensures that the CSCFG is finite.

Lemma 1. *(Finiteness) Given a specification $\mathcal{S}$, its associated CSCFG is finite.*

Proof. We show first that there do 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 the first condition of Definition 4. 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. However, the number of terms in the specification is finite and thus there is a finite number of different process calls. Moreover, every process call has only one outgoing edge as it is ensured by the second condition of Definition 4. Therefore, the number of paths is finite and the size of every path of the CSCFG is limited.

Example 4. This example makes clear the difference between the SCFG and the CSCFG, consider the specification in Fig. 3. 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 not possible control flow to `end MAIN`. However, it does not hold in the SCFG.

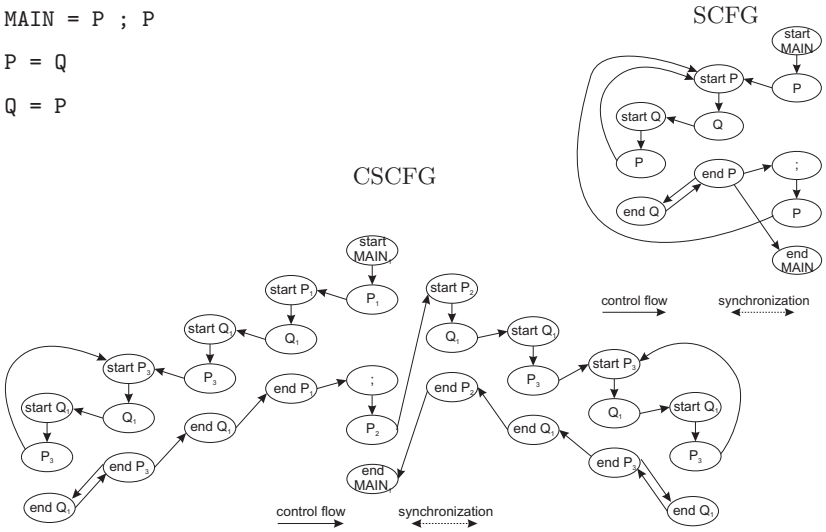

Fig. 3. SCFG and CSCFG representing an infinite computation

4 Static Slicing of CSP Specifications

We want to perform two kinds of analyses. Given an event or a process in the specification, we want, on the one hand, to determine what parts of the specification MUST be executed before (MEB) it; and, on the other hand, we want to determine what parts of the specification COULD be executed before (CEB) it.

We can now formally define our notion of slicing criterion.

Definition 5. *(Slicing Criterion) Given a specification $\mathcal{S}$, a slicing criterion is a specification position $(P, w) \in \mathcal{P}roc(\mathcal{S}) \cup \mathcal{E}vent(\mathcal{S})$.*

Clearly, the slicing criterion points to a set of nodes in the CSCFG, because the same event or process can happen in different contexts and, thus, it is represented in the CSCFG with different nodes. As an example, consider the slicing criterion (P, a) for the specification in Example 2, and observe in its CSCFG in Fig. 2 (right) that two different nodes are pointed out by the slicing criterion.

This means that a slicing criterion $\mathcal{C} = (P, w)$ is used to produce a slice with respect to *all* possible executions of w. We use function $nodes(\mathcal{C})$ to refer to all the nodes in the CSCFG pointed out by the slicing criterion $\mathcal{C}$.

Given a slicing criterion (P, w), we use the CSCFG to calculate MEB. In principle, 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 pieces of code which cannot be executed but they refer to the slicing criterion (e.g., dead code). The union of paths from `MAIN` to $nodes(\mathcal{C})$ is neither a solution because it would be too imprecise by including in the slice parts of code which are executed before the slicing criterion only in some executions. For instance, in the process `(b→a→STOP)□(c→a→STOP)`, `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 neither a solution as it can be seen in the process `a→((b→SKIP)||(c→SKIP));d` where `b` must be executed before `d`, but it does not belong to all the paths from `MAIN` to `d`.

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

Definition 6. *(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 $\mathcal{S}$; a MEB slice of $\mathcal{S}$ with respect to $\mathcal{C}$ is a subgraph of $\mathcal{G}$, $MEB(\mathcal{S}, \mathcal{C}) = (N', E'_c, E'_l, E'_s)$ with $N' \subseteq N$, $E'_c \subseteq E_c$, $E'_l \subseteq E_l$ and $E'_s \subseteq E_s$, where $N' = \{n | n \in N$ and $\forall\ X = (\texttt{MAIN} \rightarrow^* m), m \in nodes(\mathcal{C})\ .\ n \in X\}$, $E'_c = \{(n, m) | n \mapsto m \in E_c$ and $n, m \in N'\}$, $E'_l = \{(n, m) | n \rightsquigarrow m \in E_l$ and $n, m \in N'\}$ and $E'_s = \{(n, m) | n \leftrightarrow m \in E_s$ and $n, m \in N'\}$.*

Algorithm 1 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 which must be executed before it. Then, it returns MEB as the intersection of all these sets. Each set is computed with an iterative process that takes a node and (i) it follows backwards all the control-flow edges. (ii) Those nodes that could not be executed before it are added to a black list (i.e., they are discarded because they belong to a non-executed choice). And (iii) synchronizations are followed in order to reach new nodes that must be executed before it.

The algorithm always terminates. We can ensure this due to the invariant $pending \cap Meb = \varnothing$ which is always true at the end of the loop (8). Then, because Meb increases in every iteration (5) and the size of N is finite, *pending* will eventually become empty and the loop will terminate.

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 7. *(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 $\mathcal{S}$; a CEB slice of $\mathcal{S}$ with respect*

Algorithm 1 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 CSCFG's subgraph

Function $buildMeb(n)$:=
(1) $pending := \{n' \mid (n' \mapsto o) \in E_c\}$ where $o \in \{n\} \cup \{o' \mid o' \leftrightarrow n\}$
(2) $Meb := pending \cup \{o \mid o \in N$ and $\texttt{MAIN} \mapsto^* o \mapsto^* m,\ m \in pending\}$
(3) $blacklist := \{p \mid p \in N \backslash Meb$ and $o \mapsto^* p$, with $o \in choices(Meb)\}$
(4) $pending := \{q \mid q \in N \backslash (blacklist \cup Meb)$
and $q \leftrightarrow r \vee (q \rightsquigarrow r$ and $r \not\mapsto^* n)$ with $r \in Meb\}$
(5) while $\exists\ m \in pending$ do
(6) $Meb := Meb \cup \{m\} \cup \{o \mid o \in N$ and $\texttt{MAIN} \mapsto^* o \mapsto^* m\}$
(7) $sync := \{q \mid q \in N \backslash (blacklist \cup Meb)$
and $q \leftrightarrow r \vee (q \rightsquigarrow r$ and $r \not\mapsto^* n)$ with $r \in Meb\}$
(8) $pending := (pending \backslash Meb) \cup sync$
(9) return Meb

Return: $MEB(\mathcal{S}, \mathcal{C}) = (N' = \bigcap_{n \in nodes(\mathcal{C})} buildMeb(n)$,
$E'_c = \{(n,m) | n \mapsto m \in E_c$ and $n, m \in N'\}$,
$E'_l = \{(n,m) | n \rightsquigarrow m \in E_l$ and $n, m \in N'\}$,
$E'_s = \{(n,m) | n \leftrightarrow m \in E_s$ and $n, m \in N'\})$.

to $\mathcal{C}$ is a subgraph of $\mathcal{G}$, $CEB(\mathcal{S}, \mathcal{C}) = (N', E'_c, E'_l, E'_s)$ with $N' \subseteq N$, $E'_c \subseteq E_c$, $E'_l \subseteq E_l$ and $E'_s \subseteq E_s$, where $N' = \{n | n \in N$ and $\exists\ \texttt{MAIN} \rightarrow^* n \rightarrow^* m$ with $m \in nodes(\mathcal{C})\}$, $E'_c = \{(n,m) | n \mapsto m \in E_c$ and $n, m \in N'\}$, $E'_l = \{(n,m) | n \rightsquigarrow m \in E_l$ and $n, m \in N'\}$, $E'_s = \{(n,m) | n \leftrightarrow m \in E_s$ and $n, m \in N'\}$.

Therefore, $MEB(\mathcal{S}, \mathcal{C}) \subseteq CEB(\mathcal{S}, \mathcal{C})$. The graph $CEB(\mathcal{S}, \mathcal{C})$ can be computed with Algorithm 2 which, roughly, traverses the CSCFG forwards following all the paths that could be executed in parallel to nodes in $MEB(\mathcal{S}, \mathcal{C})$.

The algorithms presented can extract a slice from any specification formed with the syntax of Fig 1. However, note that only two operators have a special treatment in the algorithms: choices (because they introduce alternative computations) and synchronized parallelisms (because they introduce synchronizations). Other operators such as prefixing, interleaving or sequential composition can be treated similarly.

5 Related Work

Program slicing has been already applied to concurrent programs of different programming paradigms, see e.g. [19, 18]. As a result, different graph representations have arisen to represent synchronizations. The first proposal by Cheng [3] was later improved by Krinke [8, 9] and Nanda [13]. 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

Algorithm 2 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 CSCFG's subgraph
Initialization:

$$Ceb := \{m \mid m \in N_1 \text{ and } MEB(\mathcal{S},\mathcal{C}) = (N_1, E_{c1}, E_{l1}, E_{s1})\}$$
$$loopnodes := \{n \mid n_1 \mapsto^+ n \mapsto^* n_2 \rightsquigarrow n_3 \text{ and } (n \leftrightarrow n') \notin E_s \text{ with } n' \notin Ceb,\ n_1 \in choices(Ceb) \text{ and } n_3 \in Ceb\}$$
$$Ceb := Ceb \cup loopnodes$$
$$pending := \{m \mid m \notin (Ceb \cup nodes(\mathcal{C})) \text{ and } (m' \mapsto m) \in E_c, \text{ with } m' \in Ceb \backslash choices(Ceb)\}$$
$$\cup\{p_1 \mid p \mapsto p_1 \in E_c \text{ and } p \mapsto p_2 \in E_c \text{ and } p_1 \neq p_2, \text{ with } p \in parallel(loopnodes) \text{ and } p_2 \in loopnodes\}$$

Repeat
(1) if $\exists\ m \in pending \mid (m \leftrightarrow m') \notin E_s$ or $((m \leftrightarrow m') \in E_s$ and $m' \in Ceb)$
(2) then $pending := pending \backslash \{m\}$
(3) $Ceb := Ceb \cup \{m\}$
(4) $pending := pending \cup \{m'' \mid (m \mapsto m'') \in E_c$ and $m'' \notin Ceb\}$
(5) else if $\exists\ m \in pending$ and $\forall\ (m \leftrightarrow m') \in E_s\ .\ m' \in pending$
(6) then $candidate := \{m' \mid (m \leftrightarrow m') \in E_s\}$
(7) $Ceb := Ceb \cup \{m\} \cup candidate$
(8) $pending := (pending \backslash Ceb) \cup \{n \mid n \notin Ceb$ and $m \mapsto n\}$ $\cup\{o \mid o \notin Ceb$ and $m' \mapsto o$, with $m' \in candidate\}$

Until a fix point is reached

Return: $CEB(\mathcal{S},\mathcal{C}) = (N' = Ceb,$
$E'_c = \{(n,m) | n \mapsto m \in E_c \text{ and } n, m \in N'\},$
$E'_l = \{(n,m) | n \rightsquigarrow m \in E_l \text{ and } n, m \in N'\},$
$E'_s = \{(n,m) | n \leftrightarrow m \in E_s \text{ and } n, m \in N'\}).$

CSP, because their work is based on a different kind of synchronization. They use the following concept of *interference* to represent program synchronizations.

Definition 8. *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.

In addition, 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.

6 Implementation

We have implemented the MEB and CEB analyses and the algorithm to build the CSCFG for ProB. ProB [11] is an animator for the B-Method which also supports other languages such as CSP [1, 12]. ProB has been implemented in Prolog and it is publicly available at `http://www.stups.uni-duesseldorf.de/ProB`.

The implementation of our slicer has three main modules. The first module implements the construction of the CSCFG. Nodes and control and loop edges are built following the algorithm of Definition 4. For synchronization edges we use an algorithm based on the approach by Naumovich et al. [14]. For efficiency reasons, the implementation of the CSCFG does some simplifications which reduces 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 the graph compactation which joins together all the nodes defining a sequential path (i.e., without nodes with more than one output edge). For instance, the graph of Fig. 4 is the compacted version of the CSCFG in Fig. 2. Here, e.g., node `6` accounts for the sequence of nodes `P` $\rightarrow$ `start P`. The compacted version is a very convenient representation because the reduced data structure speeds up the graph traversal process. The second module performs the MEB and CEB analyses by

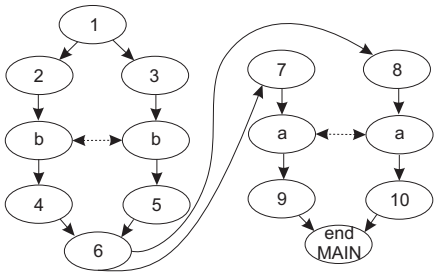

Fig. 4. Compacted version of the CSCFG in Fig. 2

implementing Algorithm 1 and Algorithm 2. Finally, the third module allows the tool to communicate with the ProB interface in order to get the slicing criterion and show the highlighted code.

We have integrated our tool into the graphical user interface of ProB. This allows us to use features such as text coloring in order to highlight the final slices. In particular, once the user has loaded a CSP specification, she can select (with the mouse) the event 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 specification position of interest. Then, the tool internally generates the CSCFG of the specification 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 5 shows a screenshot of the tool showing a slice of a CSP specification.

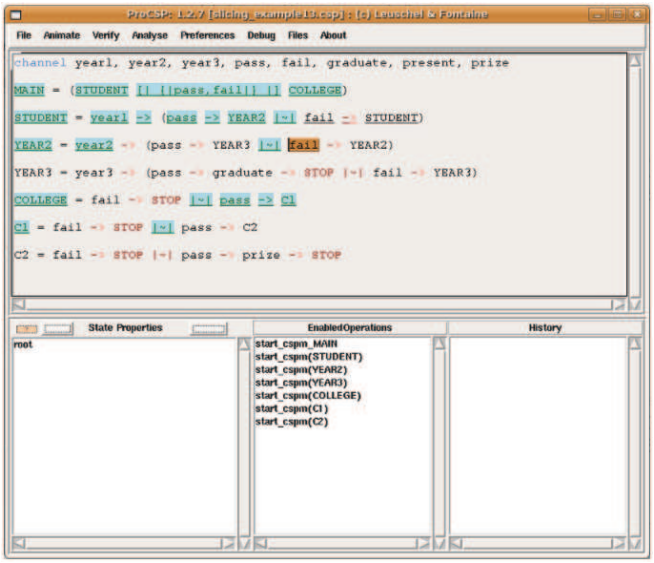

Fig. 5. Slice of a CSP specification in ProB

7 Conclusions

This work defines two new static analysis that can be applied to languages with explicit synchronizations such as CSP. 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 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 which must be executed (in any execution) before the slicing criterion (MEB analysis); and (iii) we can also produce an augmented slice which 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 which are 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 it is able to distinguish between different contexts in which a process is called.

We have built a prototype which implements all the data structures and algorithms defined in the paper; and we have integrated it into the system ProB. Preliminary experiments has demonstrated the usefulness of the technique.

References

1. Michael Butler and Michael Leuschel. Combining CSP and B for specification and property verification. In *Proceedings of Formal Methods 2005*, LNCS 3582, pages 221–236, Newcastle upon Tyne, 2005. Springer-Verlag.
2. D. Callahan and J. Sublok. Static analysis of low-level synchronization. In *In proceedings of the 1988 ACM SIGPLAN and SIGOPS workshop on Parallel and distributed debugging (PADD'88)*, pages 100–111, New York, NY, USA, 1988. ACM.
3. J. Cheng. Slicing concurrent programs - a graph-theoretical approach. In *Automated and Algorithmic Debugging*, pages 223–240, 1993.
4. 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*, 9(3):319–349, 1987.
5. M.J. Harrold, G. Rothermel, and S. Sinha. Computation of interprocedural control dependence. In *International Symposium on Software Testing and Analysis*, pages 11–20, 1998.
6. C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 26(1):100–106, 1983.
7. K. M. Kavi, F. T. Sheldon, and B. Shirazi. Reliability analysis of CSP specifications using petri nets and markov processes. In *Proc. 28th Annual Hawaii Int. Conf. on System Sciences;: Software Technology, 3-6 January 1995, Wailea, HI*, volume 2, pages 516–524, 1995.
8. J. Krinke. Static slicing of threaded programs. In *Workshop on Program Analysis For Software Tools and Engineering*, pages 35–42, 1998.
9. J. Krinke. Context-sensitive slicing of concurrent programs. In *Proc. FSE/ESEC*, pages 178–187, 2003.
10. P. Ladkin and B. Simons. Static deadlock analysis for csp-type communications, 1995.
11. Michael Leuschel and Michael J. Butler. ProB: an automated analysis toolset for the B method. *STTT*, 10(2):185–203, 2008.
12. Michael Leuschel and Marc Fontaine. Probing the depths of CSP-M: A new FDR-compliant validation tool. In *Proceedings ICFEM 2008*, LNCS. Springer-Verlag, 2008. to appear.
13. M.G. Nanda and S. Ramesh. Slicing concurrent programs. In *ISSTA '00: Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, pages 180–190, New York, NY, USA, 2000. ACM.
14. G. Naumovich and G.S. Avrunin. A conservative data flow algorithm for detecting all pairs of statements that may happen in parallel. *SIGSOFT Softw. Eng. Notes*, 23(6):24–34, 1998.
15. A. W. Roscoe, Paul H. B. Gardiner, Michael 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 *First International Workshop Tools and Algorithms for Construction and Analysis of Systems (TACAS '95)*.
16. F. Tip. A Survey of Program Slicing Techniques. *Journal of Programming Languages*, 3:121–189, 1995.
17. M.D. Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.
18. J. Zhao, J. Cheng, and K. Ushijima. Slicing concurrent logic programs, 1997.
19. Jianjun Zhao. Slicing aspect-oriented software. In *Proceedings of the 10th IEEE International Workshop on Programming Comprehension*, pages 251–260, June 2002.