An algorithm to detect synchronized events of CSP specifications*

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

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

Abstract. Concurrent programs present additional difficulties to the design and implementation of static analyses. The main reason is that the order in which the statements of the programs are executed is non-deterministic. In presence of interference between processes due to shared memory, the non-deterministic order can lead the same program either to non-termination or to a deadlock. However, this non-determinism is limited by synchronizations. Synchronizations impose some restrictions that the execution order must obey, and thus, they can be used to predict the evolution of our concurrent systems. In this paper we propose an algorithm to predict the synchronizations of a concurrent system. This algorithm can be used to improve the performance of many static analyses.

Keywords: Concurrent programming, Synchronization analysis, CSP.

1 Introduction

It is a great deal to predict all the possible pairs of events that can be executed in parallel in concurrent programs. However, the usefulness of such information is clear because a lot of analyses use it to know the behavior of programs. It could be also used in debugging programs, deadlock analysis, automatic optimization or program slicing.

One of the most studied concurrent languages is the *Communicating Sequential Processes* (CSP) [2]. It is a process algebra which allows the specification of complex systems with multiple interacting processes. The study and transformation of such systems often implies different analyses (e.g., deadlock analysis [4], reliability analysis [3], refinement checking [11], etc.). Recently, *MEB* and *CEB* (and closely related) static analyses have been proposed [6, 7, 9] for concurrent languages that allow explicit synchronizations such as CSP. These analyses are based on the *Context-sensitive Synchronized Control Flow Graph (CSCFG)*. The

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

CSCFG is a data structure used to finitely represent the (often infinite) computations of a given specification. This new kind of control flow graph has the peculiarity of being able to represent different concurrent languages.

In this article, we introduce an algorithm that predicts all the synchronizations that can occur in a specification. For the sake of concretion, we base our explanations on CSP, but, as the CSCFG is independent of the language, this work can be used in other concurrent languages.

The closest work to our proposal is the *May Happen in Parallel (MHP)* algorithm presented in [10], which has been successfully applied in some recent works. The main difference is the graph used to represent the program, since they use the *Parallel Execution Graph (PEG)*, which is essentially different to the CSCFG and whose purpose is not the detection of synchronizations but the detection of statements that can be executed concurrently.

The rest of the paper has been organized as follows. Section 2 introduces some aspects related to CSCFGs and CSP that will be used along the paper. Section 3 shows an algorithm able to detect synchronizations in a given CSP specification. Section 4 shows through an example how this algorithm works. Section 5 indicates some experimental results. Finally, Section 6 concludes and discusses some ideas for future work.

2 Preliminaries

In this section we introduce some aspects related to CSCFGs and CSP which serve as a basis for the next sections. In the following we will assume familiarity with the CSP language, and we refer the interested reader to [9] where CSP's syntax is explained.

We are going to introduce some relevant aspects of CSP needed to understand our proposal. A CSP specification is a set of definition of *processes*, where the processes are built with *calls* to processes, *events* and *operators*. Events are the most interesting element since they are the basis of synchronizations. Other interesting element is the *parallelism operator* ($||$), that introduces a bifurcation in the execution, allowing its branches to be executed in parallel. In parallelisms, a set of events which has to be synchronized is specified, in such a way that the execution is stopped in one branch or in both (representing this case a deadlock) until the same (synchronized) event occurs in both branches. Another important operator is the *choice* ($\Box$), which creates two alternative paths to be executed. Finally, the *sequential composition* (;) is used to sequentially connect two processes. Despite the technique presented in this paper supports full CSP as in [9], we only explained these operators because they are more relevant in this work. The input of our algorithm is a directed graph (the CSCFG $\mathcal{G} = (N, E_c, E_s, E_l)$) which finitely represents all possible executions of a given specification. The computation of this graph is described in [9]. It has a set of nodes N, each one has a label pointing to a position in the specification; and edges are divided in three classes:

– *Control edges* (E_c): They are directed edges, and represent the control flow of the execution (denoted with $\mapsto$).
– *Synchronization edges*(E_s): They represent the synchronizations between events. They are undirected edges (denoted with $\nleftrightarrow$).
– *Loop edges* (E_l): They represent the occurrence of an infinite loop in the execution. They are directed edges, where the last node of the loop points to the first node (denoted with $\rightsquigarrow$).

For further information about this graph, we refer the reader to [9].

3 An algorithm to detect synchronizations

In this section we present an algorithm to detect what synchronizations can occur in (all possible executions of) some specification. First of all, we have to build the CSCFG of the specification. The main function is presented as `Main_algorithm`. Roughly speaking, it first analyzes a CSCFG and adds all possible synchronizations between events of the graph. Then, it removes those parts of the graph that are non-reachable due to deadlocks. Basically, this algorithm is a loop where a parallelism node is chosen according to a top-down order. After the node selection, the algorithm calls to procedure *disconnect non-synchronized nodes* (`dnsn` for briefly) having as argument the set of nodes returned by function *build synchronization edges left* (`bse_left` for briefly). Procedure `dnsn` removes the non-reachable parts of the graph. Function `bse_left` draws all possible synchronizations of a given parallelism. The main algorithm finishes when all the parallelisms of the graph have been processed (they are included in the set *treated*). It is important to remark that the CSCFG $\mathcal{G} = (N, E_c, E_l, E_s)$ is a global variable of the whole process (it can be accessed and/or modified at any time).

```
Main_algorithm =
    treated = ∅
    repeat
        n ∈ {||}\treated | (∄n' ∈ {||}\treated.n' ≠ n ∧ n' ↦* n)
        {n1, n2} = {n' ∈ N | (n ↦ n') ∈ Ec}
        dnsn(bse_left(n1, {n2}, n))
        treated = treated ∪ {n}
    until (treated ∩ {||} = {||})
end_Main_algorithm
```

The algorithm that includes the synchronizations of the graph is divided into two functions: *build synchronization edges left* and *right* (`bse_left` and `bse_right`, respectively). Function `bse_left` is used to guide the process, and function `bse_right` is, roughly, an operator used to draw the synchronizations in the CSCFG. The parameters of `bse_left` are:

– n_{s_1}: the current node of the left branch.
– s_2: the set of current nodes of the right branch.

– $n_{||}$: the parallelism node.

Function `bse_left` returns a set of nodes which are candidates to be disconnected. First of all, this function checks whether the label of node n_{s_1} is in the set of synchronizable nodes *events* in the parallelism $n_{||}$. If it is in the set, function `bse_right` is called and, after this call, if there is not any synchronization drawn, a set containing node n_{s_1} and *candidates* is returned by function `bse_right`. Otherwise, the algorithm continues by checking wether the new set of right nodes s'_2 is empty, returning, in that case, the set *candidates*$'$. If this condition does not hold function *calculate next nodes* (function `cn`) is called in order to get the set of successor nodes s'_1 of the current node n_{s_1}. Finally, function `bse_left` returns *candidates*$'$ joined with the result of calling `bse_left` $(successor, s'_2, n_{||})$ for every successor node.

```
bse_left(n_{s_1}, s_2, n_|| = ||_{events}) =
    (s'_2, candidates') = (s_2, ∅)
    if l(n_{s_1}) ∈ events then
        (s'_2, candidates') = bse_right(S_2, n_{s_1}, n_||, λ)
        if {n_s ∈ N | ∃n' ∈ s_2.n' ↦* n_s ∧ n_s ↮ n_{s_1}} = ∅ then
            return candidates' ∪ {n_{s_1}}
        if s'_2 = ∅ then return candidates'
        (s'_1, _) = cn(n_{s_1}, n_||)
        case s'_1 of
            ∅: return candidates'
            {son}: return candidates' ∪ bse_left(son, s'_2, n_||)
            {son_1, son_2}:
                return candidates' ∪ bse_left(son_1, s'_2, n_||) ∪ bse_left(son_2, s'_2, n_||)
end_bse_left
```

We delay the explanation of function `bse_right` and explain first how function `cn` computes the successors of a node in a parallelism. Computing successor nodes is not trivial (i.e., they cannot be known just following control edges), because we have to know where the parallelism ends in order to stop searching after this node, and in order to handle loops. The function `cn` receives as parameters n, the current node, and $n_{||}$, the parallelism node. It finishes returning a set of successor nodes and a flag which indicates whether a loop has been included. First of all, the function distinguishes between three possible values (zero, one or two) of the number of n's sons. If it is zero, the node could be one with an outcoming loop edge. In this case, if there is some synchronizable event nodes of $n_{||}$ into the path from the beginning of the loop (n_l) to its end (n), a set that just includes n_l is returned, and the flag is put to *true*, represented by T. This is the unique case where this flag is put to *true*. In the rest of cases, its value will be *false*, represented by F.

```
cn(n, n_|| = ||_{events}) =
    case {n' ∈ N | (n ↦ n') ∈ E_c} of
        ∅:
            if (∃n_l ∈ N | (n ⇝ n_l) ∈ E_l) ∧ (∃n' ∈ {n_l ↦* n' ↦* n} | l(n') ∈ events)
```

```
            then return (\{n_l\}, T)
            else return (\emptyset, F)
        \{son\}:
            if (n \in \{SKIP\}) \wedge (\exists n_; \in N \mid (n \mapsto n_;) \in E_c) then
                (first_;, \_) = fsc(n_;)
                if n_{||} \in \{n' \in N \mid first_; \mapsto^* n' \mapsto^* n\} then return (\emptyset, F)
            return (\{son\}, F)
        \{son_1, son_2\}: return (\{son_1, son_2\}, F)
end_cn
```

If node n has only one son, it could be a $SKIP$ followed by a sequential composition operator $n_;$. In this case, function `fsc` is called in order to know the top most bifurcation $first_;$ between all bifurcations finishing in node $n_;$. After this call, it is checked whether the parallelism node $n_{||}$ is included in the path from node $first_;$ to node $n_;$ returning an empty set if it happens. In all other cases, the successors of node n will be returned.

```
fsc(n) =
    befores = \{n_b \in N \mid (n_b \mapsto n) \in E_c\}
    path_common = \bigcap_{n_b \in befores} \{n \in N \mid MAIN \mapsto^* n \mapsto^* n_b\}
    return (n \in path_common \mid (\nexists n' \in path_common . (n \mapsto n') \in E_c), befores)
end_fsc
```

We continue now introducing function `bse_right`. This function has the following parameters:

- s_2: the set of current nodes of the right branch.
- n_{s_1}: the current node of the left branch.
- $n_{||}$: the parallelism node.
- *loops_stack*: a stack which controls that the whole process cannot be looped. Its elements are tuples which contain a set of nodes and a flag. The set of nodes represents the available right branch's nodes when a loop is added. The flag is used to know whether a synchronization edge has been drawn before to add it again to the stack. Here, the empty stack is represented by λ.

```
bse_right(s_2, n_{s_1}, n_{||} = ||_{events}, loops_stack) =
    if s_2 = \emptyset then return (\emptyset, \emptyset)
    Given n_{s_2} \in s_2
    if l(n_{s_1}) = l(n_{s_2}) then
        if (n_{s_1} \leftrightarrow n_{s_2}) \in E_s then
            if loops_stack = \lambda then return bse_right(s_2 \backslash \{n_{s_2}\}, n_{s_1}, n_{||}, loops_stack)
            else return bse_right(s''_2, n_{s_1}, n_{||}, tail(loops_stack))
                where (s''_2, \_) = head(loops_stack)
        else
            E_s = E_s \cup \{n_{s_1} \leftrightarrow n_{s_2}\}
            return { bse_right(s_2 \backslash \{n_{s_2}\}, n_{s_1}, n_{||},
                       (s_l, T) : tail(loops_stack))          if (s_l, \_) = head(loops_stack)
                   { bse_right(s_2 \backslash \{n_{s_2}\}, n_{s_1}, n_{||}, loops_stack)  otherwise
```

```
    if (l(n_{s_2}) ∈ events) then return (s''_2, candidates'' ∪ {n_{s_2}}) where
        (s''_2, candidates'') = bse_right(s_2\{n_{s_2}}, n_{s_1}, n_||, loops_stack)
    (s'_2, is_loop) = cn(n_{s_2}, n_||)
    if s'_2 = ∅ return bse_right(s_2\{n_{s_2}}, n_{s_1}, n_||, loops_stack)
    if is_loop then
        if head(loops_stack) = (s_2, F) then return (∅, ∅)
        else return bse_right(s'_2 ∪ s_2\{n_{s_2}}, n_{s_1}, n_||, ((s_2, F) : loops_stack))
    else return bse_right(s'_2 ∪ s_2\{n_{s_2}}, n_{s_1}, n_||, loops_stack)
end_bse_right
```

The function will return the new set of right branch's nodes as well as a set of candidates nodes. If s_2 is empty two empty sets are returned. Otherwise, a node n_{s_2} of s_2 is selected. The function begins by checking whether the labels of nodes n_{s_1} and n_{s_2} are the same, meaning that there exists a synchronization between them. In that case, firstly, it has to check if the synchronization exists yet, and in that case it will return a recursive call replacing the first parameter according to the stack's value: if it is empty the value will be $s_2 \backslash \{n_{s_2}\}$, otherwise the set on the stack's head. If the synchronization has not been drawn yet, then it is included in set E_s and the function returns a recursive call where first parameter is $s_2 \backslash \{n_{s_2}\}$ and fourth the stack replacing stack head's flag by *true*. If n_{s_1} and n_{s_2} labels were different, but the label of n_{s_2} is in *events*, the function executes a recursive call with $s_2 \backslash \{n_{s_2}\}$ as first parameter and its result is returned adding n_{s_2} to its set of candidates. If none of before cases holds, successor nodes s'_2 and flag *is_loop* are calculated by means of function `cn`. If s'_2 is an empty set, the function returns a recursive call with $s_2 \backslash \{n_{s_2}\}$ as a first parameter. Otherwise, if a loop has been included by `cn`, represented with value *true* in flag *is_loop* and s_2 is in the stack's head, two empty sets are returned, meaning that a loop in the process has been detected. If s_2 is not in the head, it is returned the result of calling itself with $s'_2 \cup s_2 \backslash \{n_{s_2}\}$ as first parameter and adding to the stack the tuple $(s_2, false)$. Finally, if anything has been returned until now, the function returns a recursive call with $s'_2 \cup s_2 \backslash \{n_{s_2}\}$ as first parameter.

We have finished the presentation of the synchronization process, and now we are going to describe how works the piece of the process which is responsible to disconnect non-reachable sections of the graph. Procedure `dnsn` is the main ingredient of this process. Its parameters are: *candidates*, a set of nodes candidates to be disconnected and the parallelism node $n_{||}$. Firstly, a node n of $candidates \cap N$ is selected. This intersection assures that the node selected exists, since some nodes in *candidates* could be removed by the process yet. After selecting n, it is checked whether exists some synchronization between n and some event of its opposite branch. If there is not any synchronization, node n has to be disconnected, therefore function `dsc` is called in order to disconnect sequential compositions that could be below n. After, all the reachable nodes from node n are removed from N and all the edges that have some node not present in N are removed from each edge set. The procedure finishes with a recursive call replacing the first parameter by $candidates \backslash \{n\}$. The process follows until set *candidates* is empty.

```
dnsn(candidates, n_||) =
    if candidates = ∅ then EXIT
    Given n ∈ (candidates ∩ N)
    {n_1, n_2} = {n' ∈ N | (n_|| ↦ n') ∈ E_c}
    if (((n_1 ↦* n) ≠ ∅) ∧ ({n_s ∈ (n_2 ↦* n) | (n ↔ n_s) ∈ E_s)} = ∅)∨
        (((n_2 ↦* n) ≠ ∅) ∧ ({n_s ∈ (n_1 ↦* n) | (n ↔ n_s) ∈ E_s)} = ∅) then
        dsc(n, n)
        N = N\{n' ∈ N | n ↦* n'}
        E_c = E_c\{(n' ↦ n'') ∈ E_c | (n' ∉ N) ∨ (n'' ∉ N)}
        E_s = E_s\{(n' ↔ n'') ∈ E_c | (n' ∉ N) ∨ (n'' ∉ N)}
        E_l = E_l\{(n' ⇝ n'') ∈ E_c | (n' ∉ N) ∨ (n'' ∉ N)}
        dnsn(candidates\{n}, n_||)
end_dnsn
```

Procedure `dsc`, other important piece in the process, disconnects the sequential composition operators that have been turned non-reachable due to the disconnection of some node. According to some factors, it could remove all, some or none incoming control edges for a sequential composition node. If all incoming edges are removed, then the process has to keep disconnecting below sequential composition. The procedure receives as parameters: n, the node which has been cut and n_0, the beginning node of the backward searching. Function `cl` is used to calculate all the leafs of paths that begin in node n and finish in a sequential composition node $n_;$. Node $n_;$ has as a first node n_first, which cannot be into the path from node n to node $n_;$. Once the call is executed, if node n_first is a bifurcation, auxiliary procedure `dsc_bifurcation` is called. Otherwise, the procedure finishes.

```
dsc(n, n_0) =
    (n_;, n_first, befores, leafs) = cl(n, ∅)
    if leafs = ∅ then EXIT
    else if {n' ∈ N | (n_first ↦ n') ∈ E_c} = {n_1, n_2} then
        leafs_1 = befores ∩ {n' ∈ N | n_1 ↦* n'}
        leafs_2 = befores ∩ {n' ∈ N | n_2 ↦* n'}
        dsc_bifurcation(leafs, leafs_1, leafs_2, n_first, n_;, n, n_0)
    else EXIT
end_dsc

cl(n, visited) =
    visited' = visited ∪ {n}
    case {n' ∈ N | (n ↦ n') ∈ E_c}} of
        ∅: return (⊥, ⊥, ∅, ∅)
        {son}: if son ∈ {;} then
            (n_first, befores) = fsc(son)
            if n_first ∉ visited' then
                return (son, n_first, befores, {n})
            return cl(son, visited')
        {son_1, son_2}: return (n_;, n_first, befores, leafs_1 ∪ leafs_2) where
            (n_;, n_first, befores, leafs_i) = cl(son_i, visited') ∧ i ∈ {1, 2}
end_cl
```

The procedure `dsc_bifurcation` is the skeleton of procedure `dsc`. It has as parameters:

- $leafs$: a set of nodes representing the leafs for node n w.r.t. node $n_;$.
- $leafs_1$ and $leafs_2$: sets of nodes representing the leafs for each branch of node n_{first}.
- n_{first}: the first node of sequential composition $n_;$.
- n: the node which has been cut.
- $n_;$: the sequential composition operator node.
- n_0: the beginning node of the backward searching.

Procedure begins by checking whether set $leafs$ is equal to some of the set of leafs of node n_{first}. If it is equal to someone, there are two cases according to the type of node n_{first}. If it is a parallelism, procedure `dsc` is called with $n_;$ and n_{first} as parameters and, after, all the nodes reachable from $n_;$ are removed from N and all the incoming control edges to $n_;$ are disconnected from the sequential composition operator. Otherwise, if it is a choice, only the nodes of set $leafs$ are disconnected. If set $leafs$ was different to both sets of leafs of node n_{first}, function `lflb` is called in order to calculate the last bifurcation node n_{last} over the node n_0. The idea is to accumulate the set of leafs nodes to be disconnected from node $n_;$. If node n_{last} is a parallelism, a recursive call is made adding to the set of nodes $leafs$ the leafs of node n_{last}. If it is a choice, then the nodes in $leafs$ are disconnected from the sequential composition operator node $n_;$.

```
dsc_bifurcation(leafs, leafs1, leafs2, n_first, n_;, n, n0) =
    if (leafs = leafs1) ∨ (leafs = leafs2) then
        if n_first ∈ {||, |||} then
            dsc(n_;, n_first)
            N = N\{n' ∈ N | n_; ↦* n'}
            E_c = E_c\{(n' ↦ n_;) ∈ E_c | n' ∈ N}
        else E_c = E_c\{(n' ↦ n_;) ∈ E_c | n' ∈ leafs}
    else
        (n_last, leafs'1, leafs'2) = lflb(n0, n_;)
        if n_last ∈ {||, |||} then
            dsc_bifurcation(leafs', leafs1, leafs2, n_first, n_;, n, n0) where
                leafs' = leafs ∪ leafs'1 ∪ leafs'2
        else E_c = E_c\{(n' ↦ n_;) ∈ E_c | n' ∈ leafs}
end_dsc_bifurcation

lflb(n, n_;) =
    Given b ∈ {b' ∈ N | (b' ↦ n) ∈ E_c}
    if {son1, son2} = {n' ∈ N | (b ↦ n') ∈ E_c} then
        (n1, _, _, leafs1) = cl(son1, ∅)
        (n2, _, _, leafs2) = cl(son2, ∅)
        return (b, leafs1, leafs2)
    else return lflb(b, n_;)
end_lflb
```

4 An Example

In this section we show through an example how the whole algorithm works. We were interested in show a lot of features in only one example, then the specification that we are going to use is hard to explain, but the CSCFG which is shown will faithfully help to understand its behavior. The specification is the following.

Example 1.

$$
\begin{aligned}
&MAIN = (P \;||_{\{a,b\}}\; Q); a \rightarrow STOP \\
&P = (R \;||_{\{c,d\}}\; c \rightarrow d \rightarrow c \rightarrow d \rightarrow STOP) \;\Box \\
&\qquad (\; (a \rightarrow b \rightarrow a \rightarrow (b \rightarrow SKIP \;\Box\; b \rightarrow SKIP); \\
&\qquad\quad (b \rightarrow (a \rightarrow SKIP \;\Box\; b \rightarrow SKIP)\;) \\
&Q = (\; c \rightarrow (a \rightarrow b \rightarrow SKIP \;||_{\{a,b\}}\; a \rightarrow b \rightarrow SKIP) \;\Box \\
&\qquad\quad (a \rightarrow b \rightarrow b \rightarrow SKIP)\;); \\
&\qquad (a \rightarrow (b \rightarrow SKIP \;||_{\{a,b\}}\; a \rightarrow SKIP)) \\
&R = ((c \rightarrow SKIP) \;\Box\; (d \rightarrow SKIP)); R
\end{aligned}
$$

This specification has a parallelism between two processes P and Q. Note that process P has a loop caused by the call to R and a deadlock occurs in one of the parallelisms of Q. This deadlock is the cause why it is not possible to arrive never to the second process of the sequential composition of $MAIN$. The CSCFG for the specification of Example 1 is shown in Figure 1. It is going to serve as an input parameter for the whole process. The deadlock mentioned before happens in node 71 and the loop is into nodes 4 and 13. We can see in Figure 2 the result after applying procedure `Main_algorithm` over the graph of Figure 1. In this graph we can see with dotted lines the synchronizations built by the algorithm, as well as the cut performed. We can see, for instance, how the nodes of the events into the loop (6 and 9) have been correctly synchronized. On the other hand, if we follow the right branch of the choice of node 2, we can see how the first two events have three synchronizations. This is due to the fact that, until then, both branches of choice numbered 47 can be executed. But after this, node 65 has been removed since it is an event b, and the node 27 is an event a, then it produces a deadlock. However, the execution could continue if left branch of choice 47 is chosen. But in parallelism of node 71, the graph finishes because the deadlock into nodes 72 and 75. As a result of this we have that nodes 30 and 33 have not any possible node to be synchronized, letting node 29 as the last node of that branch. The rest of the graph is also removed, since it is impossible to be executed following CSP semantics.

5 Experiments

We have implemented the algorithm presented here for CSP. All code of the algorithm has been summarized in the functions and procedures of Section 3.

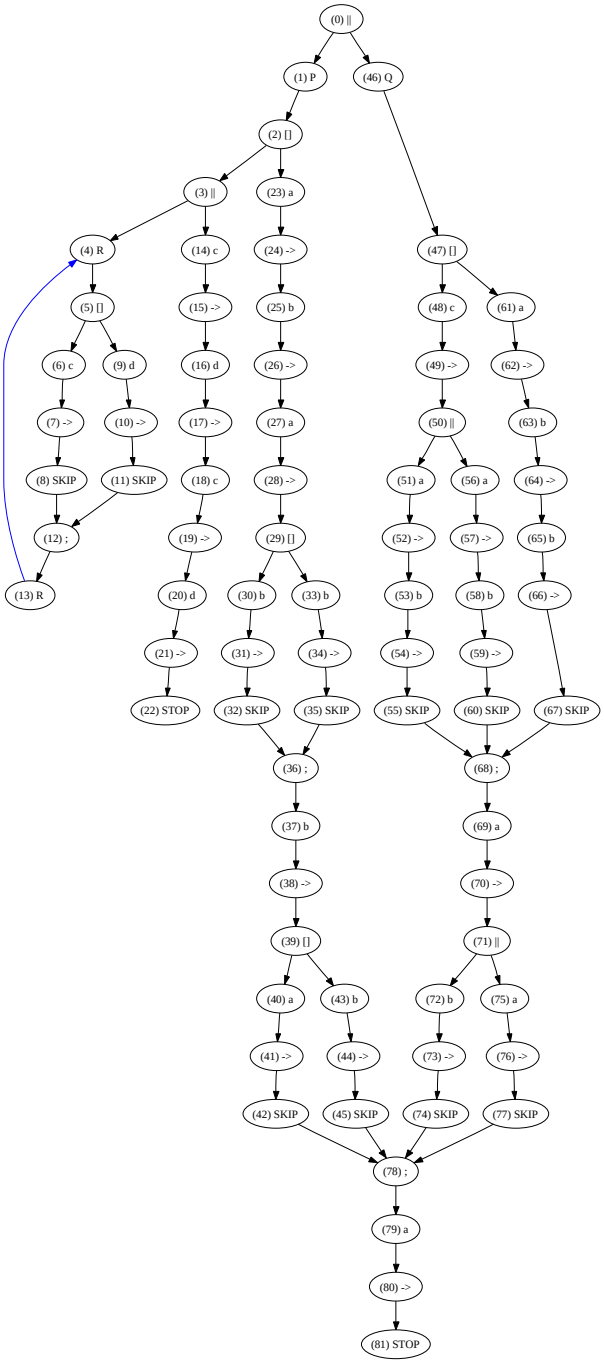

Fig. 1. CSCFG of Example 1

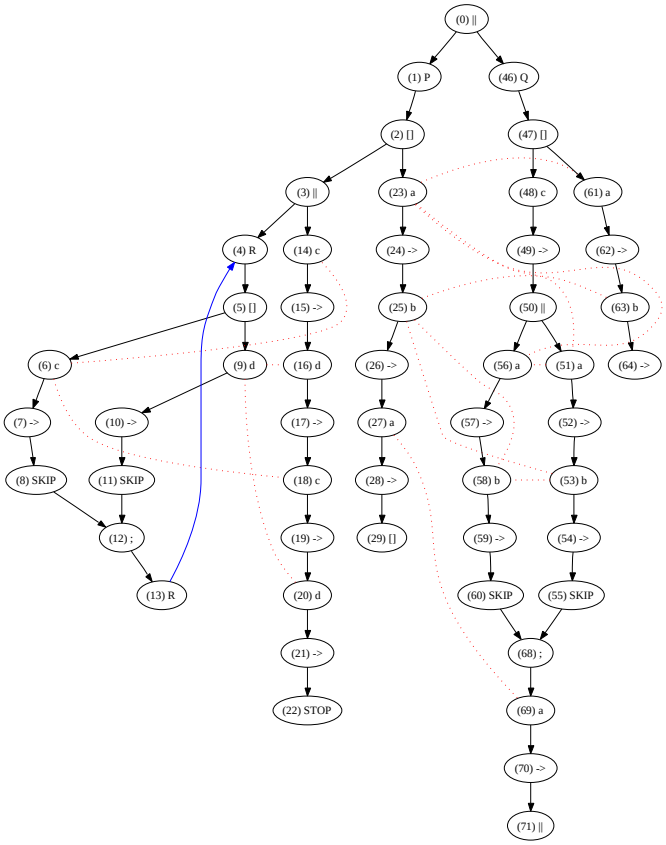

Fig. 2. CSCFG of Example 1 after applying `Main_algorithm`

The implementation is a tool called *SOC* which stands for *Slicing Of CSP* [8]. This tool has been integrated in the system ProB [5, 1]. We have opened the source code; therefore, all the implementation (both source code and executable) is publicly available. To download the tool and some other materials visit:

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

6 Conclusions and Future Work

In this paper we present a technique to calculate synchronizations for CSP specifications represented with CSCFGs. The technique has been successfully used in a slicing tool for CSP [8]. The technique takes a CSCFG and adds all possible synchronizations between the events of the CSCFG. Then, by using the synchronizations, the CSCFG is purged so that non-reachable nodes are removed. The way in that the algorithm is described throughout this paper makes it sufficiently implementation-oriented as to be adapted to any concurrent language in which

the CSCFG can be used. For future work, we plan to improve the efficiency, which is maybe the main drawback of this approach. Also, we are interested in studying the level of adaptability of our technique to other concurrent languages.

We have conducted a number of experiments, and they give an idea of its applicability to real-world specifications. It works fast for small to medium specifications; and for large specifications it is only usable if the number of synchronizable nodes is small. Otherwise, the algorithm takes long to compute all synchronizations. We have been improving the implementation of this algorithm since its first version, and we have got great time reductions during this time. This improvements have been obtained thanks to the optimization of the internal data structures.

We plan the creation of a new data structure which will be an abstraction of the CSCFG having only the information needed by the algorithm (synchronizable nodes, parallelisms, choices and sequential compositions). This idea could result in an improvement in time and in an important simplification of the whole algorithm. Hence improving the scalability of the algorithm.

References

1. 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 upon Tyne, 2005.
2. C. A. R. Hoare. Communicating sequential processes. *Communications ACM*, vol. 26(1), pp. 100–106, 1983.
3. 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.
4. P. Ladkin and B. Simons. Static deadlock analysis for csp-type communications. *Responsive Computer Systems (Chapter 5)*, Kluwer Academic Publishers, 1995.
5. 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.
6. M. Leuschel, M. Llorens, J. Oliver, J. Silva, and S. Tamarit. Static slicing of CSP specifications. In *Proceedings of the 18th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'08)*, pp. 141–150, 2008.
7. M. Leuschel, M. Llorens, J. Oliver, J. Silva, and S. Tamarit. The MEB and CEB static analysis for CSP specifications. Technical report, Department of Computer Science, Technical University of Valencia. Accessible via `http://www.dsic.upv.es/~jsilva`, Valencia, Spain, October 2008.
8. 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.
9. Leuschel, M., M. Llorens, J. Oliver, J. Silva, and S. Tamarit, "The MEB and CEB Static Analysis for CSP Specifications", In Post-proceedings of the 18th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'08), Revised Selected Papers, LNCS 5438, Springer-Verlag, pp. 103–118, 2009.

10. 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.
11. 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.