An Algorithm to Generate the Context-sensitive Synchronized Control Flow Graph*

M. Llorens, J. Oliver, J. Silva and S. Tamarit
Departamento de Sistemas Informáticos y Computación,
Universidad Politécnica de Valencia
Valencia, Spain
{mllorens,fjoliver,jsilva,stamarit}@dsic.upv.es

ABSTRACT

The verification of industrial systems specified with CSP often implies the analysis of many concurrent and synchronized components. The cost associated to these analyses is usually very high, and sometimes prohibitive, due to the complexity imposed by the non-deterministic execution order of processes and to the restrictions imposed on this order by synchronizations. To overcome this problem, there has been a recent proposal that allows to statically simplify a specification before the analyses. This simplification allows to drastically reduce the time needed by the analyses because it reduces the state explosion. Unfortunately, the approach has been implemented but it has not been formalized neither proved correct. In this paper, we formally define the data structures needed to automatically simplify a CSP specification and we define an algorithm able to automatically generate these data structures.

Categories and Subject Descriptors

D.3.3 [**Software**]: Programming Languages—*Language Constructs and Features (concurrent programming structures)*; D.1.3 [**Software**]: Programming Techniques—*concurrent programming*

General Terms

Algorithms, Languages

Keywords

CSP, CSCFG

*This work has been partially supported by the Spanish *Ministerio de Ciencia e Innovación* under grant TIN2008-06622-C03-02, by the *Generalitat Valenciana* under grant ACOMP/2009/017, and by the *Universidad Politécnica de Valencia* (Programs PAID-05-08 and PAID-06-08). Salvador Tamarit was partially supported by the Spanish MICINN under FPI grant BES-2009-015019.

SAC'10 March 22-26, 2010, Sierre, Switzerland.

1. INTRODUCTION

The *Communicating Sequential Processes* (CSP) [1, 11] 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 [4], reliability analysis [2], refinement checking [10], etc.) which are often based on a data structure able to represent all computations of a specification.

The ProB system [5, 6] is an automated analysis tool set that allows the specification, animation and verification of complex CSP systems. Due to parallelism, non-deterministic execution order, deadlocks, synchronizations, and non-terminating processes, the analysis of these systems is a very complex and costly task.

A recent work [8] proposed to simplify the process specifications before the analysis in such a way that (i) all parts of a specification which are not needed in the analysis are skipped; and (ii) the resultant specification is made executable by a transformation. This work is based on a data structure called *Context-sensitive Synchronized Control Flow Graph* (CSCFG for short). This data structure allows us to finitely represent all (often infinite) computations of a given CSP specification. Once constructed, the CSCFG is very useful to determine what parts of a specification are related to some (other) part of it. In fact, there are already defined and implemented analysis methods based on the CSCFG. See, for instance, the MEB and CEB analyses [8].

Surprisingly, the construction of a CSCFG has not been formally defined yet, and thus, it is not possible to prove correctness of the tools that work with CSCFGs (i.e., current implementations are made ad-hoc). In this work, we formally define the notion of CSCFG and present an algorithm to automatically generate the CSCFG of a CSP specification.

The rest of the paper has been organized as follows. Section 2 recalls CSP syntax. In Section 3 we formally define the CSCFG. Then, Section 4 introduces an algorithm able to generate the CSCFG associated to a CSP specification. Finally, Section 5 concludes.

2. THE SYNTAX OF CSP

In order to make the paper self-contained, this section recalls CSP's syntax.

Figure 1 summarizes the syntax constructions used in 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 x must happen before expression $Proc$.
Input. It is used to receive a message from another process. Message u is received through channel c; then process $Proc$ 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 $Proc$ is executed.
Internal choice. The system chooses (e.g., non-deterministically) 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.
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 (successfully) finishes, the second starts.
Hiding. Process $Proc$ 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 $Proc$ is executed with a set of renamed events specified with the 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 us to continue the next sequential process.
Stop. It finishes the current process; but it does not allow the next sequential process to continue.

$$
\begin{array}{llll}
 & & & \textit{Domains} \\
 & & & P, Q, R \ldots \in \mathcal{P} \text{ (processes)} \\
 & & & a, b, c \ldots \in \Sigma \text{ (events)} \\
 & & & u, v, w \ldots \in \mathcal{V} \text{ (variables)} \\
S & ::= & D_1 \ldots D_m & \text{(entire specification)} \\
D & ::= & P = Proc & \text{(process definition)} \\
Proc & ::= & Q & \text{(process call)} \\
 & | & x \rightarrow Proc & \text{(prefixing)} \\
 & | & c?u \rightarrow Proc & \text{(input)} \\
 & | & c!u \rightarrow Proc & \text{(output)} \\
 & | & Proc_1 \sqcap Proc_2 & \text{(internal choice)} \\
 & | & Proc_1 \,\Box\, Proc_2 & \text{(external choice)} \\
 & | & Proc_1 \,|||\, Proc_2 & \text{(interleaving)} \\
 & | & Proc_1 \,||_{\{\overline{x_n}\}}\, Proc_2 & \text{(synchronized parallelism)} \\
 & | & Proc_1 \,;\, Proc_2 & \text{(sequential composition)} \\
 & | & Proc \backslash \{\overline{x_n}\} & \text{(hiding)} \\
 & | & Proc[\![f]\!] & \text{(renaming)} \\
 & | & SKIP & \text{(skip)} \\
 & | & STOP & \text{(stop)}
\end{array}
$$

where $\overline{x_n} = x_1, \ldots, x_n$, $x_i \in \Sigma \cup \mathcal{V}$ and $f : \Sigma \rightarrow \Sigma$

Figure 1: Syntax of CSP specifications

EXAMPLE 1. *Consider the following specification where the labels of the terms (in grey color) can be ignored for the time being.*

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

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

Due to the choice operator this specification can produce two different sequences of events. If the left-hand side branch of the choice is selected, then the resultant sequence of events is {b} *(i.e., event* a *is never executed because it is never synchronized, hence a deadlock happens). Contrarily, if the right-hand side branch of the choice is selected, then the resultant sequence of events is* {a} *and the whole system successfully terminates.*

To illustrate the possible executions of this example, the reader can observe the CSCFG associated to this specification which is shown in Figure 2. This graph will be explained in detail in next section.

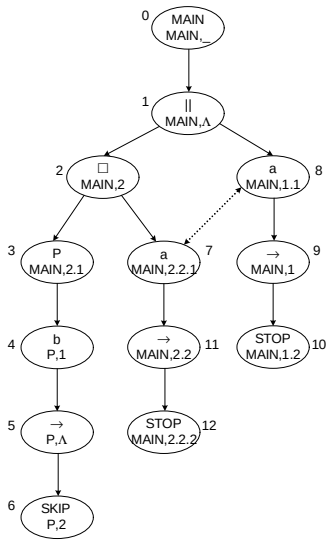

Figure 2: CSCFG associated to the specification of Example 1

3. CONTEXT-SENSITIVE SYNCHRONIZED CONTROL FLOW GRAPHS

One of the most important static analysis tools is the *Control Flow Graph* (CFG). Unfortunately, this data structure cannot represent multiple threads and, thus, it can only be used with sequential programs. In fact, for complex languages (such as CSP), being able to represent multiple threads is a necessary but not a sufficient condition. For instance, the *threaded Control Flow Graph* (tCFG) [3] can represent multiple threads; but it does not handle synchronizations between threads. A recent version of the CFG able to represent threads, synchronizations, and which is context sensitive (different process calls have different representations), the CSCFG, has been proposed [7]. However, there does not exist a formal definition of the CSCFG that relates this data structure with the execution of programs it

represents. In this section we formally define the CSCFG based on the well-known concept of control flow using CSP notation. Firstly, we need to introduce some preliminary definitions.

We use labels (that we call *specification positions*) to identify each term in a specification. Formally,

Definition 1. *(Position, specification position) Given a specification $\mathcal{S}$ with processes $\mathcal{P}$, the* positions *in $\mathcal{S}$ are represented by a sequence of natural numbers, where Λ denotes the empty sequence. They are used to address subexpressions of an expression viewed as a tree. A* specification position *is a pair (P, w) where $P \in \mathcal{P}$ and w is a position. We let $\mathcal{P}os(e)$ denote the specification position of a CSP expression e. Given a process definition $P = rhs(P)$, the specification position of its left-hand side $\mathcal{P}os(P) = (P, \_)$, and*

$$\begin{array}{l} \textit{if } rhs(P) = Q \textit{ then } \mathcal{P}os(Q) = (P, \Lambda) \textit{ for all process call } Q \\ \textit{if } rhs(P) = Proc\ op \textit{ then } \mathcal{P}os(op) = (P, \Lambda) \ \ \forall\ op \in \{\setminus, [\![]\!]\} \\ \qquad\qquad\qquad\qquad \wedge\ \ \mathcal{P}os(Proc) = (P, 1) \\ \textit{if } rhs(P) = Proc_1\ op\ Proc_2 \\ \qquad \textit{then } \mathcal{P}os(op) = (P, \Lambda) \ \ \forall\ op \in \{\rightarrow, \sqcap, \Box, |||, ||, ;\} \\ \qquad\qquad \wedge\ \ \mathcal{P}os(Proc_1) = (P, 1) \wedge \mathcal{P}os(Proc_2) = (P, 2) \end{array}$$

and, inductively, for all position w and a (sub)expression e of a process P,

$$\begin{array}{l} \textit{if } \ e = Proc\ op\ \wedge\ \mathcal{P}os(op) = (P, w)\ \forall\ op \in \{\setminus, [\![]\!]\} \\ \qquad \textit{then } \mathcal{P}os(Proc) = (P, w.1) \\ \textit{if } \ e = Proc_1\ op\ Proc_2 \\ \quad \wedge\ \mathcal{P}os(op) = (P, w)\ \forall\ op \in \{\rightarrow, \sqcap, \Box, |||, ||, ;\} \\ \qquad \textit{then } \mathcal{P}os(Proc_1) = (P, w.1)\ \wedge\ \mathcal{P}os(Proc_2) = (P, w.2) \end{array}$$

We also define the specification position of a CSP expression e as its root position, i.e., $\mathcal{P}os(Proc\ op) = \mathcal{P}os(op)$ and $\mathcal{P}os(Proc_1\ op\ Proc_2) = \mathcal{P}os(op)$.

For instance, in Example 1 each term of the specification has been labeled with its specification position (in grey color). Notice that the specification positions of each process can be viewed as a tree whose root is Λ.

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

Definition 2. *(Control-flow) Given a CSP specification $\mathcal{S}$ and two specification positions n, n' in $\mathcal{S}$, we say that the* control *of n can pass to n' iff*

- $n = P \wedge n' = first((P, \Lambda))$ *with* $rhs(P) \in \mathcal{S}$
- $n \in \{\sqcap, \Box, |||, ||\} \wedge n' \in \{first(n.1), first(n.2)\}$
- $n \in \{\rightarrow, ;\} \wedge n' = first(n.2)$
- $n = n'.1 \wedge n' = \rightarrow$
- $n \in last(n'.1) \wedge n' = \ ;$
- $n \in \{\setminus, [\![]\!]\} \wedge n' = first(n.1)$

where $first(n)$ is the specification position of the subprocess denoted by n which must be executed first:

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

and where $last(n)$ is the set of all possible termination points of the subprocess denoted by n:

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

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

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

Definition 3. *(Path) A* path *between two nodes n_1 and m, denoted by $Path(n_1, m)$, 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 $n_i \neq n_j$.*

Definition 4. *(Context) Given a labeled graph $G = (N, E_c)$ and a node $n \in N$, the* context *of n, $\mathcal{C}on(n) = \{m \mid l(m)$ is a process call and exists a loop-free path $\pi = m \mapsto^* n$ with $last(m) \notin \pi\}$.*

Intuitively speaking, the context of a node represents the set of processes in which a particular node is being executed. This is represented by the set of process calls in the computation that have not finished yet. For instance, in Figure 2 $\mathcal{C}on(4) = \{0, 3\}$, i.e., `b` is being executed after having called processes `MAIN` and `P`.

Notice that focussing on a process call node we can use the context to identify loops; i.e., we have a loop whenever $P \in \mathcal{C}on(P)$.

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

- *E_c is a set of one-way edges (denoted with $\mapsto$) representing the possible control-flow between two nodes. Control edges do not form loops. The root of the tree formed by E_c is the root position of the main process.*
- *E_l is a set of one-way edges (denoted with $\rightsquigarrow$) such that $(n_1 \rightsquigarrow n_2) \in E_l$ iff $l(n_1)$ and $l(n_2)$ are (possibly different) process calls that refer to the same process $P \in \mathcal{P}$ and $n_2 \in \mathcal{C}on(n_1)$.*
- *E_s is a set of two-way edges (denoted with $\nleftrightarrow$) representing the possible synchronization of two (event) nodes.*

Given a CSCFG, every node labeled (P, Λ) has one and only one incoming edge in E_c; and every process call node has one and only one outgoing edge which belongs to either E_c or E_l.

The CSCFG associated to the specification in Example 1 is shown in Figure 2. Each process call is connected (with a control-flow edge) to a subtree which contains the right-hand side of the called process. Each subtree is a new subgraph except if a loop is detected. In Example 1 there are no loop edges; there are only control-flow edges and one synchronization edge between nodes $(\texttt{MAIN}, 2.2.1)$ and $(\texttt{MAIN}, 1.1)$ representing the synchronization of event $\texttt{a}$.

Loop edges allow us to finitely represent infinite computations. As shown in the next example, they are used when the same process call appears twice in a path starting from MAIN, and the first process has not been terminated.

EXAMPLE 2. *Consider the following CSP specification:*

$$\texttt{MAIN} = \texttt{a}_{(\texttt{MAIN},1.1)} \rightarrow {}_{(\texttt{MAIN},1)}\texttt{a}_{(\texttt{MAIN},1.2.1)} \rightarrow {}_{(\texttt{MAIN},1.2)}\texttt{STOP}_{(\texttt{MAIN},1.2.2)} \underset{\{\texttt{a}\}}{\|}{}_{(\texttt{MAIN},\Lambda)}\texttt{P}_{(\texttt{MAIN},2)}$$
$$\texttt{P} = \texttt{a}_{(\texttt{P},1)} \rightarrow {}_{(\texttt{P},\Lambda)}\texttt{P}_{(\texttt{P},2)}$$

This specification can produce the sequence of events $\{\texttt{a}, \texttt{a}\}$ *and its associated CSCFG is shown in Figure 3, where there are a loop edge from node 8 to node 2 and two synchronization edges between nodes 4 and 3 and nodes 6 and 3.*

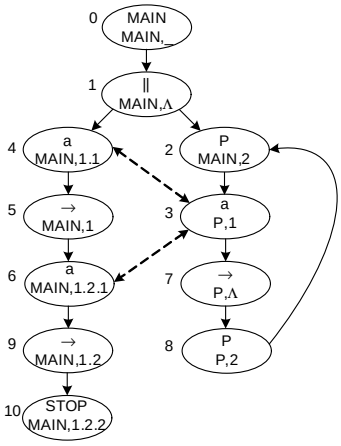

Figure 3: CSCFG of the specification in Example 2

4. AN ALGORITHM TO GENERATE THE CSCFG

This section introduces an algorithm able to generate the CSCFG associated to a CSP specification. The basic idea of the algorithm is to generate different portions of the CSCFG that correspond to different feasible executions and join them together. The algorithm uses an instrumented big-step operational semantics of CSP which generates as a side-effect the CSCFG associated to the computation performed with the semantics. The algorithm controls that the semantics is executed repeatedly in order to execute all parts of the specification (i.e., to explore all non-deterministic choices) and thus, it can join all the CSCFGs produced in order to construct a complete CSCFG for the whole specification. The instrumentation of the semantics controls that infinite computations are stopped by using the loop edges introduced in Definition 5. We have published this instrumented semantics as a technical report [9].

For the purposes of this paper, the instrumented semantics is not needed (the interested reader is referred to [9] for a detailed explanation of this semantics). In the following we can assume that we have available a function Sem that takes a *state* s as input and produces a state s' as output (i.e., $Sem(s) = s'$).

A state of the semantics is a tuple (E, G, S), where E is the expression to be evaluated (e.g., MAIN), G is a directed graph (i.e., the CSCFG constructed so far) where $\emptyset$ represents the empty graph, and S is the stack (a list of specification positions where the empty stack is denoted by $\emptyset$) which represents the branches of choices executed so far.

Given a derivation of the semantics $Sem((\texttt{MAIN}, G, S)) = (E, G', S')$, G is the initial graph, and G' is G augmented with the part of the CSCFG produced by the derivation from MAIN to E. S is used to break the indeterminism and it allows the semantics to know what branch should be selected in every choice. This is very important to ensure that the semantics does not repeat computations, and it produces a new part of the graph in every execution. S' stores the set of branches of choices that has been executed in the derivation.

The instrumented semantics evaluates the initial expression according to the CSP' standard semantics [11] except by the fact that elections in choices are determined by S. The algorithm forces the semantics to execute the branches of choices in depth-first order. For this purpose, the stack S contains a collection of pairs (p, d) where p is a specification position referring to a choice, and d is a character that can be 'l' or 'r' to indicate that the left, respectively right, branch of this choice should be explored.

Algorithm 1 generates the CSCFG associated to a given CSP specification. Essentially, this algorithm executes the semantics once and again with the cumulated CSCFG until all branches of choices are explored. The algorithm uses function Ψ to control the parts of the specification already executed. When all branches of choices have been explored (i.e., $\Psi(S) = \emptyset$), the algorithm ends.

Algorithm 1 Algorithm to generate a CSCFG

Build the initial state of the semantics: $state = (\texttt{MAIN}, \emptyset, \emptyset)$
repeat
  $(\_, G, S) = Sem(state)$
  if $\Psi(S) \neq \emptyset$ **then**
    $state = (\texttt{MAIN}, G, \Psi(S))$
  end if
until $\Psi(S) = \emptyset$
return G

where function Ψ is used to avoid already explored branches of choices:

$$\Psi(S) = \begin{cases} (\alpha, r) : S' & \text{if } S = (\alpha, l) : S' \\ \Psi(S') & \text{if } S = (\alpha, r) : S' \\ \emptyset & \text{otherwise} \end{cases}$$

EXAMPLE 3. *Consider the following specification:*

$$\texttt{MAIN} = (\texttt{a}_{(\texttt{MAIN},1.1)} \rightarrow {}_{(\texttt{MAIN},1)}\texttt{STOP}_{(\texttt{MAIN},1.2)}) \; \square_{(\texttt{MAIN},\Lambda)} ((\texttt{b}_{(\texttt{MAIN},2.1.1)} \rightarrow {}_{(\texttt{MAIN},2.1)}\texttt{STOP}_{(\texttt{MAIN},2.1.2)}) \; \square_{(\texttt{MAIN},2)} (\texttt{c}_{(\texttt{MAIN},2.2.1)} \rightarrow {}_{(\texttt{MAIN},2.2)}\texttt{STOP}_{(\texttt{MAIN},2.2.2)}))$$

Due to the choice operators this specification can produce three different sequences of events, namely $\{\texttt{a}\}$, $\{\texttt{b}\}$ *and*

$\{\mathtt{c}\}$. *The algorithm executes the semantics with the initial state* $(\mathtt{MAIN}, \emptyset, \emptyset)$. *Then, the semantics executes the first branch of the choice with specification position* $(\mathtt{MAIN}, \Lambda)$ *producing the final state* $(\mathtt{STOP}, G, [((\mathtt{MAIN}, \Lambda), \mathtt{l})])$. *Next, the right branch of the same choice is executed producing the state* $(\mathtt{STOP}, G', [((\mathtt{MAIN}, \mathtt{2}), \mathtt{l}), ((\mathtt{MAIN}, \Lambda), \mathtt{r})])$. *In a third iteration, the second branch of the choices with specification positions* $(\mathtt{MAIN}, \Lambda)$ *and* $(\mathtt{MAIN}, \mathtt{2})$ *are selected producing the final state* $(\mathtt{STOP}, G'', [((\mathtt{MAIN}, \mathtt{2}), \mathtt{r}), ((\mathtt{MAIN}, \Lambda), \mathtt{r})])$. *In the last iteration,* $\Psi([((\mathtt{MAIN}, \mathtt{2}), \mathtt{r}), ((\mathtt{MAIN}, \Lambda), \mathtt{r})]) = \emptyset$ *and thus the algorithm terminates with the final output* G'' *which is the CSCFG of the specification. Figure 4 shows the CSCFG generated by the Algorithm 1. In this CSCFG, white nodes were generated in the first iteration (i.e., they correspond to* G*); grey nodes were generated in the second iteration (i.e., they correspond to* G'*); and black nodes were generated in the third iteration (i.e., they correspond to* G''*).*

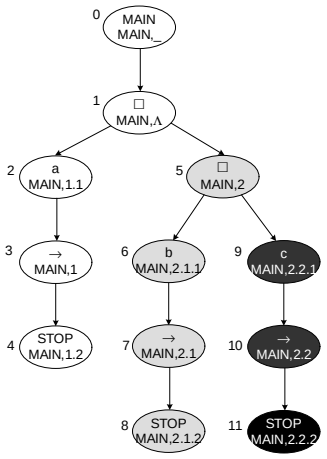

Figure 4: CSCFG associated to the specification of Example 3

5. CONCLUSIONS

This work formally defined the CSCFG and it introduced an algorithm to build the CSCFG associated to a CSP specification. We focussed on the CSP language, but the ideas presented and the algorithm itself can work with any concurrent and explicitly synchronized language by changing the semantics used (function *Sem*). The algorithm uses an instrumentation of the standard CSP's operational semantics to explore all possible computations of a specification. The semantics is deterministic because the branch to explore in every choice is predefined by the initial configuration. Therefore, the algorithm can execute the semantics several times to iteratively explore all branches and hence, generate the whole CSCFG. The CSCFG is generated even for non-terminating specifications due to the use of loop edges. The formalization of the CSCFG and the algorithm presented are an interesting result because they can serve as a reference mark to prove properties such as completeness of static analyses based on the CSCFG.

The algorithm presented explicitly relates the semantics of CSP with the construction of the CSCFG. This result constitutes the basis for the formal verification of the recent algorithms and implementations of the ProB system, because it allows to prove properties of the CSCFG and the algorithms designed to traverse it. In particular, the MEB and CEB analyses have been implemented and integrated in ProB as a debugging and program specialization tool. With the presented formalization, it is possible to formally relate these analyses with the semantics of CSP. Therefore, they could be used to automatically simplify a CSP specification by skipping those parts of its associated CSCFG not related to the property being studied. This is essential to reduce the time needed by costly analyses such as model checking and deadlock analysis.

6. REFERENCES

[1] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[2] K. M. Kavi, F. T. Sheldon, B. Shirazi, and A. R. Hurson. Reliability analysis of CSP specifications using Petri nets and Markov processes. In *Proc. 28th Annual Hawaii Intl. Conf. on System Sciences (HICSS'95)*, vol. 2 (Software Technology), pp. 516–524, Kihei, Maui, Hawaii, USA, 1995.

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

[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*, volume 10(2), pages 185–203, 2008.

[6] M. Leuschel and M. Fontaine. Probing the depths of CSP-M: A new FDR-compliant validation tool. *Proc. 10th Intl. Conf. on Formal Engineering Methods (ICFEM'08)*, volume 5256 of *LNCS*, pages 278–297, Springer-Verlag, 2008.

[7] M. Leuschel, M. Llorens, J. Oliver, J. Silva, and S. Tamarit. SOC: a slicer for CSP specifications. *Proc. 2009 ACM SIGPLAN Symp. on Partial Evaluation and Semantics-based Program Manipulation (PEPM'09)*, pages 165–168, Savannah, GA, USA, 2009.

[8] M. Leuschel, M. Llorens, J. Oliver, J. Silva, and S. Tamarit. The MEB and CEB static analysis for CSP specifications. *Post-proceedings of the 18th Intl. Symp. on Logic-Based Program Synthesis and Transformation (LOPSTR'08)*, Revised Selected Papers, volume 5438 of *LNCS*, Springer-Verlag, pages 103–118, 2009.

[9] M. Llorens, J. Oliver, J. Silva, and S. Tamarit. Semantics-based CSCFG generation. Technical report DSIC-II/05/09, Department of Information Systems and Computation, Technical University of Valencia. Accessible via `http://www.dsic.upv.es/~jsilva`, Valencia, Spain, October 2009.

[10] 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. *Proc. of the First Intl. Workshop Tools and Algorithms for Construction and Analysis of Systems (TACAS'95)*, pages 133–152, 1995.

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