SOC: A Slicer for CSP Specifications[*]

[System Demonstration]

Michael Leuschel
Institut für Informatik
Universitätsstrasse 1, D-40225
Düsseldorf, Germany
leuschel@cs.uni-duesseldorf.de

Marisa Llorens
Technical Univ. of Valencia
Camino de Vera S/N E-46022
Valencia, Spain
mllorens@dsic.upv.es

Javier Oliver
Technical Univ. of Valencia
Camino de Vera S/N E-46022
Valencia, Spain
fjoliver@dsic.upv.es

Josep Silva
Technical Univ. of Valencia
Camino de Vera S/N E-46022
Valencia, Spain
jsilva@dsic.upv.es

Salvador Tamarit
Technical Univ. of Valencia
Camino de Vera S/N E-46022
Valencia, Spain
stamarit@dsic.upv.es

ABSTRACT

This paper describes *SOC*, a program slicer for CSP specifications. In order to increase the precision of program slicing, *SOC* uses a new data structure called *Context-sensitive Synchronized Control Flow Graph* (CSCFG). Given a CSP specification, *SOC* generates its associated CSCFG and produces from it two different kinds of slices; which correspond to two different static analyses. We present the tool's architecture, its main applications and the results obtained from experiments conducted in order to measure the performance of the tool.

Categories and Subject Descriptors

F.3.1 [**Theory of Computation**]: Logics and meaning of programs—*specifying and verifying and reasoning about programs*
; D.3.1 [**Software**]: Programming Languages—*formal definitions and theory*

General Terms

Languages, Theory

Keywords

Program slicing, Software engineering

[*]This work has been partially supported by the EU (FEDER) and the Spanish MEC/MICINN under grants TIN2005-09207-C03-02, TIN2008-06622-C03-02, and *Acción Integrada* HA2006-0008.

1. INTRODUCTION

Program slicing is a well-known technique to extract the part of a program which is related to some point of interest known as slicing criterion. It was first proposed as a debugging technique [9] to allow a better understanding of the portion of code which revealed an error; nowadays, it has been successfully applied to a wide variety of software engineering tasks, such as program understanding, debugging, testing, specialization, etc. (See [8, 1] for a survey).

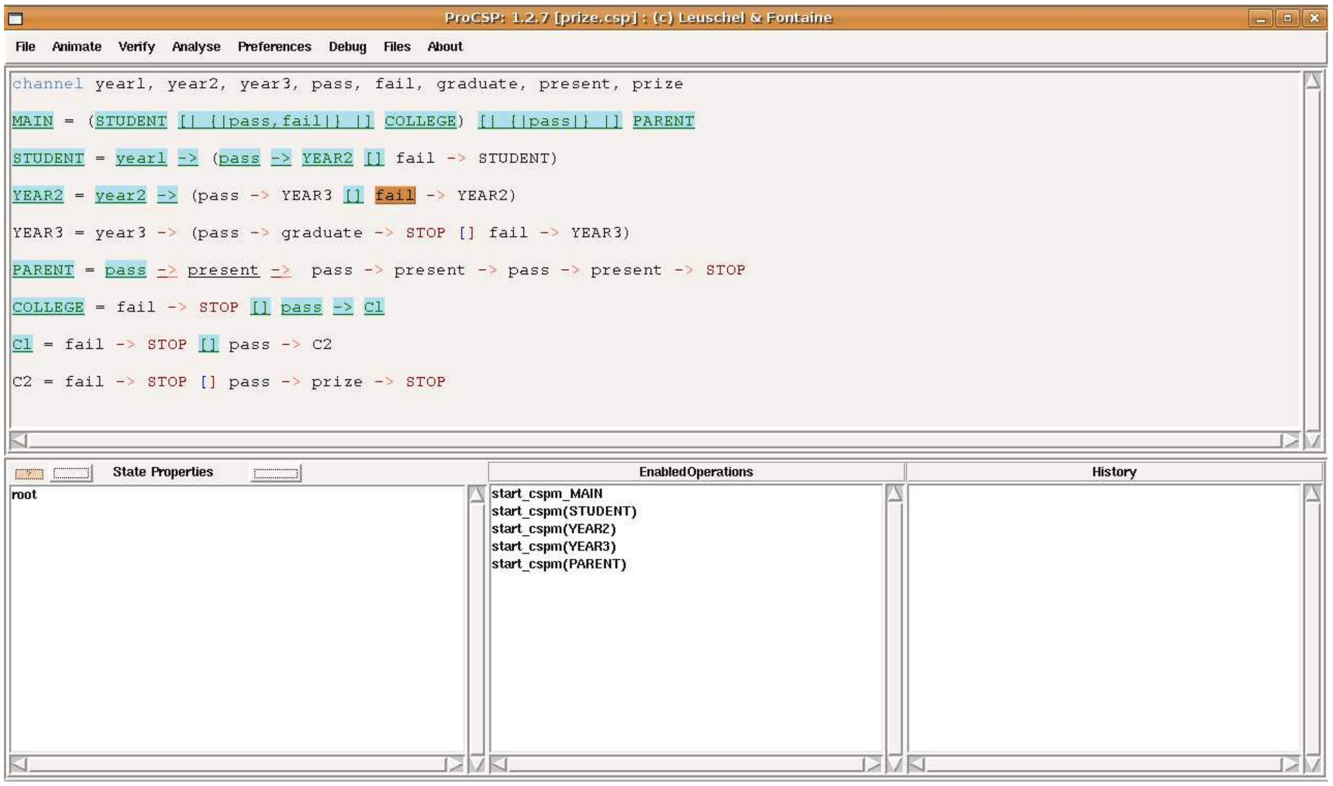
Recently, two new static analysis based on program slicing has been proposed in the context of concurrent and explicitly synchronized languages [6]. The first analysis is called MEB (which stands for *Must be Executed Before*), and it allows us to extract those parts of a specification that must be executed (in any execution) before a given point (thus they are an implicit precondition). The second analysis is called CEB (which stands for *Could be Executed Before*), and it allows us to extract those parts of a specification that could (in some execution), and could not, be executed before a given point.

This paper describes the implementation of both techniques for the *Communicating Sequential Processes* (CSP) [4] language. *SOC* has been integrated in the system ProB [5, 2] a CSP development environment and animator. Our experiments have demonstrated the usefulness of the tool with three main clear applications: debugging, program comprehension and program simplification (the tool can be used as a preprocessing stage of other analyses and/or transformations in order to reduce the complexity of the CSP specification). A clear advantage of *SOC* is that it relays on the construction of an internal data structure which is language-independent. Therefore, *SOC* could be easily adapted to other languages.

In Section 2 we show the applications of this tool and an example of use. In Section 3 we describe the architecture of *SOC*. Finally, in Section 4 we show a summary of some experiments which show the speedup and performance of our tool.

2. *SOC* IN PRACTICE

In this section, we describe the purpose of our tool and


ProCSP: 1.2.7 [prize.csp] : (c) Leuschel & Fontaine
File Animate Verify Analyse Preferences Debug Files About
State Properties
EnabledOperations
History

```
channel year1, year2, year3, pass, fail, graduate, present, prize

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

STUDENT = year1 -> (pass -> YEAR2 [] fail -> STUDENT)

YEAR2 = year2 -> (pass -> YEAR3 [] fail -> YEAR2)

YEAR3 = year3 -> (pass -> graduate -> STOP [] fail -> YEAR3)

PARENT = pass -> present -> pass -> present -> pass -> present -> STOP

COLLEGE = fail -> STOP [] pass -> C1

C1 = fail -> STOP [] pass -> C2

C2 = fail -> STOP [] pass -> prize -> STOP
```

```
root
```

```
start_cspm_MAIN
start_cspm(STUDENT)
start_cspm(YEAR2)
start_cspm(YEAR3)
start_cspm(PARENT)
```

Figure 1: Slice of a CSP specification produced by *SOC*

how it can be used to extract slices from CSP specifications. Let us consider the following example to show the usefulness of the technique.

EXAMPLE 2.1. *Consider the CSP specification*[1] *of Figure 1. In this specification we have three processes (*STUDENT*,* PARENT *and* COLLEGE*) executed in parallel and synchronized on common events. Process* STUDENT *represents the three-year academic courses of a student; process* PARENT *represents the parent of the student who gives her a present when she passes a course; and process* COLLEGE *represents the college who gives a prize to those students which finish without any fail.*

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 highlighted parts. This is called MEB analysis. Therefore,* SOC *is a powerful tool for program comprehension. Note, for instance, that in order to fail in the second year, the student has necessarily passed the first year. But, the parent could or could not give a present to his son (indeed if he passed the first year) because this specification does not force the parent to give a present to his son until he has passed the second year. This is not so obvious from the specification, and* SOC *can help to understand the real meaning of the specification.*

We can additionally be interested in knowing what parts could be executed before the same event. This is called CEB analysis. In this case, our technique adds to the slice the underscored parts because they could be executed (in some executions) before the marked event. This can be useful, e.g., for debugging. If the slicing criterion is an event that executed incorrectly (i.e., it should not happen in the execution), then the slice produced contains all the parts of the specification which could produce the wrong behavior.

A third application of our tool is program specialization. SOC *is able to extract executable slices with a program transformation applied to the generated slices. The specialized specification contains all the necessary parts of the original specification whose execution leads to the slicing criterion (and then, the specialized specification finishes).*

*Note that, in the slices produced by both analyses in Figure 1, the slice produced could be made executable by replacing the removed parts by "*STOP*" or by "$\longrightarrow$* STOP*" if the removed expression has a prefix.*

As described in the previous example, the slicing process is completely automatic. 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 source position of interest. Then, the tool internally generates an internal data structure which represents all possible computations, and uses the MEB and CEB algorithms to construct the slices. The result is shown to the user by highlighting the part of the specification that must (respectively could) be executed before the specified event.

There is another application of *SOC* which was our origi-

[1]We refer those readers non familiarized with CSP to, e.g., [4].

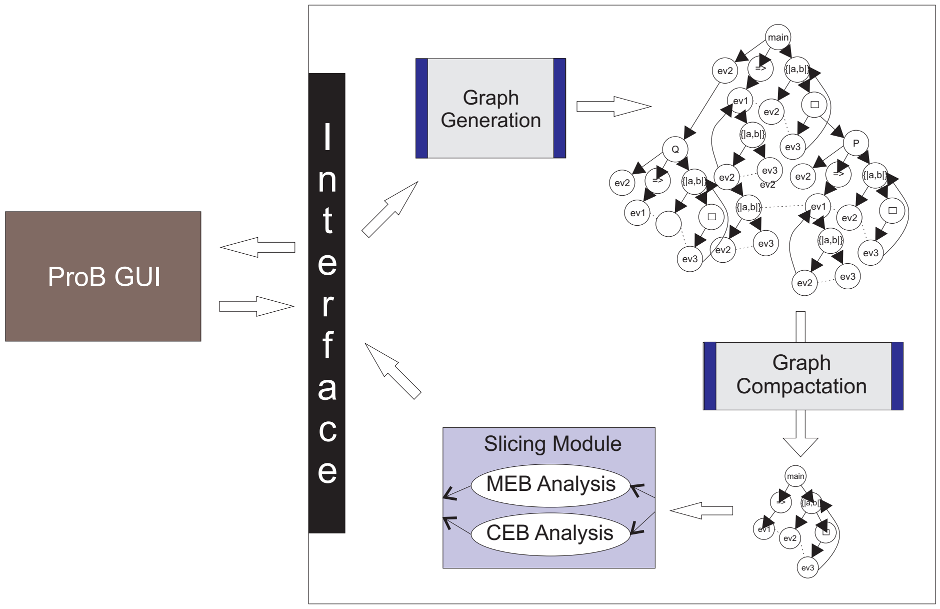


Figure 2: Slicer's Architecture

nal aim when we developed this tool. ProB is able to perform different static analyses over CSP specifications. However, due to the complexity of the specifications and to the parallel and non-deterministic execution of processes, these analyses usually become too costly as to be used with real programs. *SOC* can be used as a preprocessing stage of these analyses in order to reduce the size of the specification and, thus, the size of the data structures used in, and the complexity of, the static analyses.

The technical details of this slicing technique, including the algorithm for the MEB and CEB analyses, and the internal data structure used—the *context-sensitive synchronized control flow graph*—can be found in [7].

3. ARCHITECTURE

In this section, we describe the internal structure of *SOC*. ProB [5] is an animator for the B-Method which also supports other languages such as CSP [2]. ProB has been implemented in Prolog and it is publicly available at `http://www.stups.uni-duesseldorf.de/ProB`.

SOC has been implemented in Prolog and it has been integrated in ProB. Therefore, *SOC* can take advantage of ProB's graphical features to show slices to the user. In order to be able to color parts of the code, it has been necessary to implement the source code positions detection; in such a way that ProB can color every subexpression which is sliced by *SOC*. Apart from the interface module for the communication with ProB, *SOC* has three main modules which we describe in the following:

Graph Generation

The usual data structure for program slicing (the *system dependence graph*) cannot be used in CSP. In contrast, it is needed a new data structure called *Context-sensitive Synchronized Control Flow Graph* (CSCFG) as described in [7]. Therefore, the first task of the slicer is to build a CSCFG. The module which generates the CSCFG from the source program is the only module which is ProB dependent. This means that *SOC* could be used in other systems by only changing the graph generation module.

Graph Compactation

The original definition of the CSCFG is inaccurate from an implementation point of view. Therefore, we have implemented a module which reduces the size of the CSCFG by removing unnecessary nodes and by joining together those nodes that form paths that the slicing algorithms must traverse in all cases. This compactation not only reduces the size of the stored CSCFG, but it also speeds up the slicing process due to the reduced number of nodes to be processed.

Slicing Module

This is the main module of the tool. It is further composed of two submodules which implement the algorithms to perform the MEB and CEB analyses on the compacted CSCFGs. Depending on the analysis selected by the user this module extracts a subgraph from the compacted CSCFG using either MEB or CEB. Then, it extracts from the subgraph the part of the source code which forms the slice. If the user has selected to produce an executable slice, then the slice is transformed to become executable (it mainly fills gaps in the produced slice in order to respect the syntax of the language). The final result is then returned to ProB in such a way that ProB can either highlight the final slice or save a new CSP executable specification in a file.

Figure 2 summarizes the internal architecture of *SOC*. Note that both the graph compactation module and the slicing module take a CSCFG as input, and hence, they are independent of ProB.

4. BENCHMARKING THE SLICER

In order to measure the performance and the slicing capabilities of our tool, we conducted some experiments over a subset of the examples listed in

Table 1: Benchmark time results

benchmark	CSCFG	MEB Analysis	CEB Analysis	Total
ATM.csp	1239 ms.	7083 ms.	438 ms.	8760 ms.
RobotControlling.csp	586 ms.	923 ms.	2175 ms.	3684 ms.
Buses.csp	11 ms.	17 ms.	2 ms.	30 ms.
Prize.csp	23 ms.	116 ms.	17 ms.	156 ms.
Phils.csp	39 ms.	11 ms.	152 ms.	202 ms.
TrafficLights.csp	245 ms.	115 ms.	631 ms.	991 ms.
Processors.csp	7 ms.	7 ms.	7 ms.	21 ms.
ComplexSynchronization.csp	1365 ms.	98107 ms.	250 ms.	99722 ms.
Computers.csp	40 ms.	426 ms.	11 ms.	477 ms.
Highways.csp	4555 ms.	92 ms.	40 ms.	4687 ms.

Table 2: Benchmark size results

benchmark	Ori_CSCFG	Com_CSCFG	MEB Slice	CEB Slice	(%)
ATM.csp	9855 bytes	7533 bytes	332 bytes	448 bytes	34.94 %
RobotControlling.csp	19103 bytes	8738 bytes	309 bytes	1053 bytes	240.78 %
Buses.csp	2112 bytes	1572 bytes	114 bytes	114 bytes	0.00 %
Prize.csp	4534 bytes	3606 bytes	83 bytes	94 bytes	13.25 %
Phils.csp	10209 bytes	4417 bytes	80 bytes	594 bytes	642.50 %
TrafficLights.csp	11587 bytes	7668 bytes	110 bytes	578 bytes	425.45 %
Processors.csp	1683 bytes	1035 bytes	78 bytes	91 bytes	16.67 %
ComplexSynchronization.csp	13338 bytes	10780 bytes	634 bytes	717 bytes	13.09 %
Computers.csp	3361 bytes	2521 bytes	260 bytes	260 bytes	0.00 %
Highways.csp	7450 bytes	5642 bytes	85 bytes	151 bytes	77.65 %

`http://www.dsic.upv.es/~jsilva/soc/examples`.
For each benchmark, Table 1 summarizes the time spent to generate the compacted CSCFG (this includes the generation plus the compactation phases), to produce the MEB and CEB slices, and the total time. Table 2 summarizes the size of all objects participating in the slicing process: Column `Ori_CSCFG` shows the size of the CSCFG of the original program. Column `Com_CSCFG` shows the size of the compacted CSCFG. Columns `MEB Slice` and `CEB Slice` show respectively the size of the MEB and CEB slices. Finally, column `(%)` shows the difference in size between the MEB and CEB slices. Clearly, CEB slices are always equal or greater than their MEB counterparts.

The CSCFG compactation technique has not been published. We have implemented it in our tool and the experiments show that the size of the original specification is substantially reduced using this technique. The size of both MEB and CEB slices obviously depends on the slicing criterion selected. This table compares both slices with respect to the same criterion and, therefore, gives an idea of the difference between them.

All the information related to the experiments, the source code of the benchmarks, the slicing criteria used, the source code of the tool and other material can be found at:
`http://www.dsic.upv.es/~jsilva/soc`.

5. ACKNOWLEDGMENTS

We want to thank Mark Fontaine for his help in the implementation. He adapted the ProB module which detects source code positions.

6. REFERENCES

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

[2] M. Butler and M. 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.

[3] 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.

[4] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 26(1):100–106, 1983.

[5] M. Leuschel and M. J. Butler. ProB: an automated analysis toolset for the B method. *STTT*, 10(2):185–203, 2008.

[6] M. Leuschel, M. Llorens, J. Oliver, J. Silva, and S. Tamarit. Static Slicing of CSP Specifications. *Proc. of the 18th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'08)*, pages 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] F. Tip. A Survey of Program Slicing Techniques. *Journal of Programming Languages*, 3:121–189, 1995.

[9] M. Weiser. *Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method*. PhD thesis, The University of Michigan, 1979.

APPENDIX

A. OUTLINE OF DEMO PRESENTATION

This appendix summarizes the system demonstration we will present in case of acceptance. Firstly, we will explain the motivation of our tool *SOC*, and we will present the theoretical framework needed to contextualize the technical background of this tool. Secondly, we will describe *SOC*'s implementation by discussing its architecture and how it has been integrated in the system ProB. Next, we will show how *SOC* can be used to extract slices from CSP specifications following an example. In this part we will show the applications of *SOC*. Finally, we will show a summary of some experiments which show the speedup and performance of the tool.

A.1 Introduction

SOC is a program slicing tool for *Communicating Sequential Processes* (CSP) [4] specifications. Program slicing [1] 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*. We use slicing in order to perform static analysis in concurrent and explicitly synchronized languages. In particular, we focuss on the CSP language. CSP allows the programmer to specify complex systems with multiple interacting processes. Figure 3 summarizes the syntax constructions used in our CSP specifications.

Our technique uses program slicing to extract the part of a CSP specification which is related to a given event (the slicing criterion) in the specification. In particular, we are interested in performing two kinds of analyses. Given an event or a process in the CSP 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. 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. Its usefulness can be easily shown with an example.

Example A.1. *Consider the specification shown in Figure 1. In this specification there are 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.*

Let us assume that 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 highlighted 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.*

As it is usual in static analysis, we need a data structure able to finitely represent the (often infinite) computations of our specifications. We adapt for CSP a data structure [3] introduced by Callahan and Sublok, the *Synchronized Control Flow Graph* (SCFG), which explicitly represents synchronizations between threads with a special edge for synchronization flows. We proposed in [7] 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. Intuitively speaking, the context of a node represents the set of processes in which a particular node is being executed. 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 source 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.

We can see the difference between the SCFG and the CSCFG with an example.

Example A.2. *Figure 4 shows the SCFG and the CSCFG—for the time being, the reader can ignore the color distinction of nodes—of the following specification:*

MAIN = (P $||_{\{b\}}$ Q) ; (P $||_{\{a\}}$ R)
P = a $\rightarrow$ b $\rightarrow$ SKIP
Q = b $\rightarrow$ c $\rightarrow$ SKIP
R = d $\rightarrow$ a $\rightarrow$ SKIP

Each node is labeled with the source position it represents. The SCFG represents every source position with a single node. In contrast, the CSCFG provides a different representation for each context in which a procedure call is made. This can be seen in Figure 4 (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 source position) and they belong to the same loop. This property ensures that the CSCFG is finite.*

Given a slicing criterion (a set of nodes in the CSCFG), we use the CSCFG to calculate MEB and CEB analyses. The MEB analysis computes, for each node in the slicing criterion, 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.

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

Figure 3: Syntax of CSP specifications

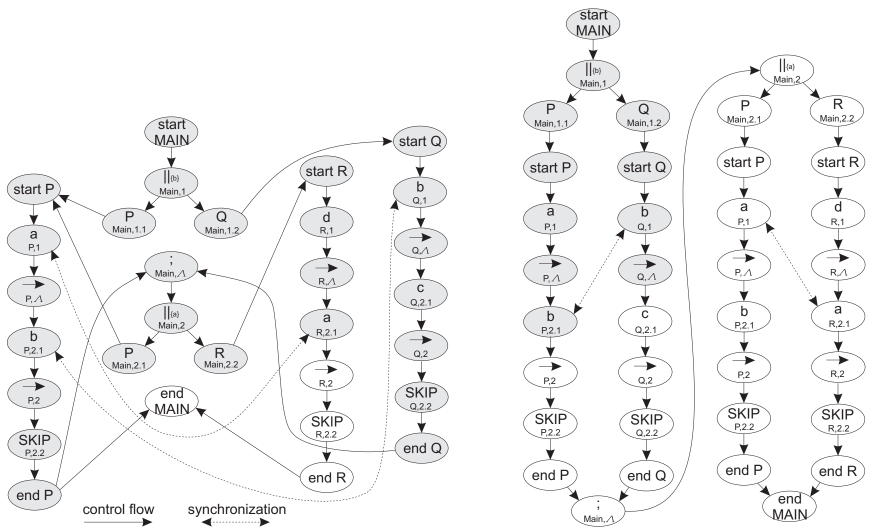


Figure 4: SCFG (left) and CSCFG (right) of the program in Example A.2

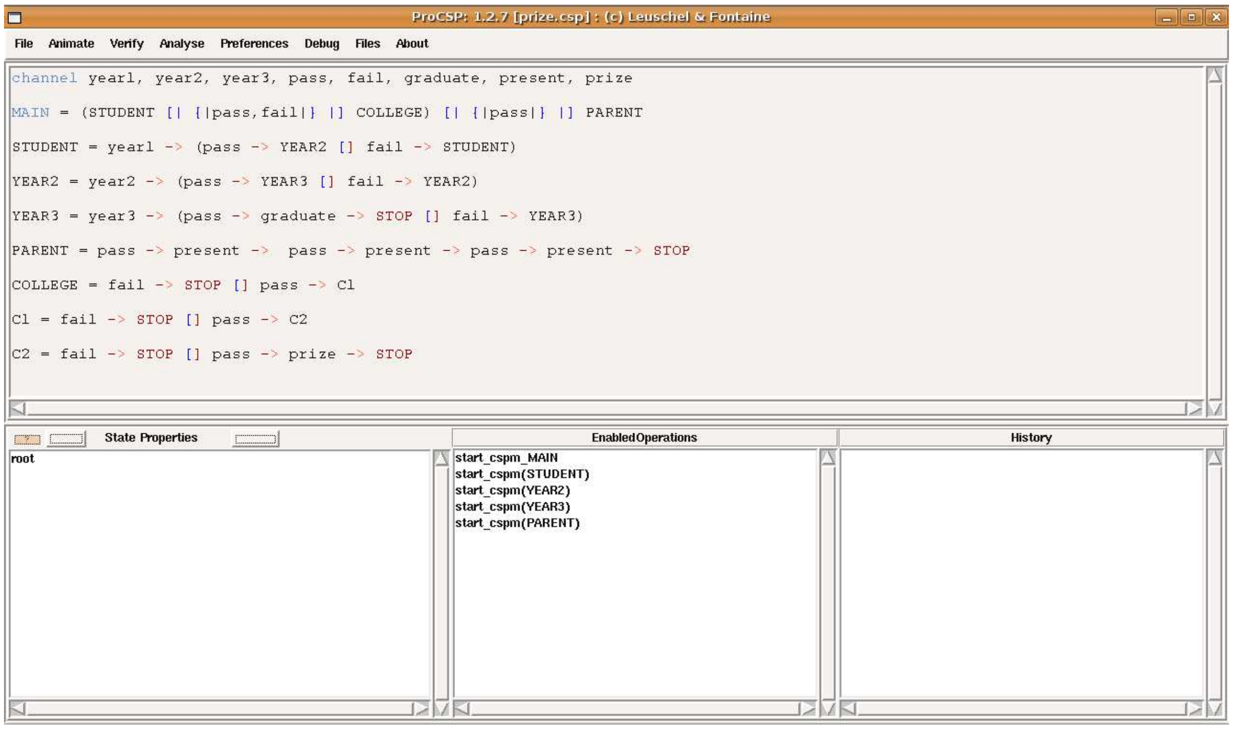
The CEB analysis computes the set of nodes in the CSCFG that could be executed before a given node n. This means that all those nodes that must be executed before n are included, but also those nodes that are executed before n in some executions, and they are not in other executions (e.g., due to non synchronized parallelism). The MEB slices obtained from the CSCFGs of Figure 4 are colored in gray. Note that the CSCFG is much more precise than the SCFG.

A.2 *SOC*'s Architecture

SOC is a slicer for CSP specifications which implements both MEB and CEB analyses. Our experiments have demonstrated the usefulness of the tool with three main clear applications: debugging, program comprehension and program simplification. In particular, using *SOC* for program simplification allows to reduce the complexity of CSP specifications in such a way that other analyses can be applied to the simplified specification. Therefore, *SOC* can be used as a preprocessing stage of other analyses and/or transformations.

SOC has been integrated in the system ProB [5, 2], a CSP development environment and animator for the B-Method which also supports other languages such as CSP. ProB has been implemented in Prolog and it is publicly available at `http://www.stups.uni-duesseldorf.de/ProB`. ProB enables a user to animate a specification, either interactively or automatically. ProB was developed using SICStus Prolog [13]. It uses Tcl/Tk for the Graphical User Interface (a Java version is also available) and dot/dotty from the Graphviz package. In Figure 5 we show the graphical user interface of ProB. The menu bar contains the various commands to access the features of ProB. It includes the `File` menu, with a submenu `Recent Files` to quickly access the files previously opened in ProB. Notice the two couples of commands `Open/Save` and `Reopen/Save and Reopen`, the latter reopening the currently opened file and reinitialising completely the state of the animation and the model checking processes. The `About` menu provides help on the tool, including a command to check if an update is available on the ProB website. By default, ProB starts with a limited set of commands in the `Beginner` mode. The `Normal` mode gives access to more features and can be set in the menu `Preferences` → `User Mode`. Under the menu bar, the main window contains four panes:

- In the top pane, the specification of the machine is displayed with syntax highlight, and can also be edited by typing directly in this pane;
- At the bottom, the animation window is composed of three panes which display, at the current point during the animation:
  – The current state of the machine (`State Properties`), listing the current values of the machine


ProCSP: 1.2.7 [prize.csp] : (c) Leuschel & Fontaine
File Animate Verify Analyse Preferences Debug Files About
State Properties
EnabledOperations
History

```
channel year1, year2, year3, pass, fail, graduate, present, prize

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

STUDENT = year1 -> (pass -> YEAR2 [] fail -> STUDENT)

YEAR2 = year2 -> (pass -> YEAR3 [] fail -> YEAR2)

YEAR3 = year3 -> (pass -> graduate -> STOP [] fail -> YEAR3)

PARENT = pass -> present ->  pass -> present -> pass -> present -> STOP

COLLEGE = fail -> STOP [] pass -> C1

C1 = fail -> STOP [] pass -> C2

C2 = fail -> STOP [] pass -> prize -> STOP
```

```
root
```

```
start_cspm_MAIN
start_cspm(STUDENT)
start_cspm(YEAR2)
start_cspm(YEAR3)
start_cspm(PARENT)
```

Figure 5: Graphical user interface of ProB

variables;

– The enabled operations (`Enabled Operations`), listing the operations whose preconditions and guards are true in this state;

– The history of operations that leaded to this state (`History`).

The animation facilities of ProB allow users to gain confidence in their specifications. These features are user-friendly as the user does not have to guess the right values for the operation arguments or choice variables, and she uses the mouse to operate the animation. At each point during the animation process, several useful commands displaying various information on the machine are available in the `Analyse` menu.

SOC has been implemented in Prolog and it has been integrated in ProB. Therefore, *SOC* can take advantage of ProB's graphical features to show slices to the user. In order to be able to color parts of the code, it has been necessary to implement the source code positions detection; in such a way that ProB can color every subexpression which is sliced by *SOC*. Apart from the interface module for the communication with ProB, *SOC* has three main modules which we can see in Figure 2.

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

Graph compactation. The original definition of the CSCFG is inaccurate from an implementation point of view. Therefore, we have implemented a module which reduces the size of the CSCFG by removing unnecessary nodes and by joining together those nodes that form a paths that the slicing algorithms must traverse in all cases. This compactation not only reduces the size of the stored CSCFG, but it also speeds up the slicing process due to the reduced number of nodes to be processed. For instance, the graph of Figure 6 is the compacted version of the CSCFG in Figure 4.

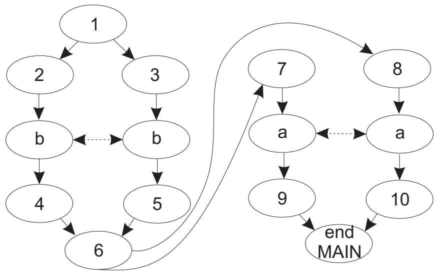


Figure 6: Compacted version of the CSCFG in Figure 4

Slicing module. This is the main module of the tool. It is further composed of two submodules which implement the algorithms to perform the MEB and CEB analyses on the compacted CSCFGs. Depending on the analysis selected by the user this module extracts a subgraph from the compacted CSCFG using either MEB or CEB. Then, it extracts from the subgraph the part of the source code which forms the slice. If the user has selected to produce an executable slice, then the slice is transformed to become executable (it mainly fills gaps in the produced slice in order to respect the syntax of the language). The final result is then returned to ProB in such a way that ProB can either highlight the final slice or save a new CSP executable specification in a file.

A clear advantage of *SOC* is that almost all the modules of the tool are language-independent. They relay on the construction of the CSCFG of the program. Therefore, *SOC* could be easily adapted to other languages by only modifying the module which constructs the CSCFG.

A.3 *SOC* in Practice

Let us consider the example A.1 to show how *SOC* can be used to extract slices from CSP specifications. The user has two possibilities: to edit the specification directly in the top pane or to load it from a file (if it exists as a previously edited file). Once the program is loaded, the user can slice it with a process which is fully automatic. Concretely, the user can select (with the mouse) the event or process call she is interested in. This simple action is enough to define a slicing criterion because the tool can automatically determine the process and the source position of interest. In our example, the slicing criterion is event `fail` of process `YEAR2`, as we can see in Figure 7.

Then, she selects command `Highlight Slice` from `Analyse` → `Slicing` menu. The tool internally generates the CSCFG of the specification (saved in a file .dot) 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 1 shows a screenshot of ProB showing a slice of our CSP specification example w.r.t. the slicing criterion `(YEAR2,fail)`. We can observe highlighted in green the MEB slice and underlined the CEB slice which coincide with the expected results.

Finally, the user can view the generated CSCFG opening the corresponding .dot file. In Figure 8, the CSCFG generated when `Analyse` → `Slice` is selected is shown. The nodes of the MEB slice are dark.

A.4 Benchmarking the slicer

In order to measure the specialization capabilities of our tool, we conducted some experiments over a subset of the examples listed in
`http://www.dsic.upv.es/~jsilva/soc/examples`.

Results are summarized in Table 1 and Table 2. For each benchmark, Table 1 summarizes the time spent to generate the compacted CSCFG (this includes the generation plus the compactation phases), to produce the MEB and CEB slices, and the total time. Table 2 summarizes the size of all objects participating in the slicing process: Column `Ori_CSCFG` shows the size of the CSCFG of the original program. Column `Com_CSCFG` shows the size of the compacted CSCFG. Columns `MEB Slice` and `CEB Slice` show respectively the size of the MEB and CEB slices. Finally, column `(%)` shows the difference in size between the MEB and CEB slices. Obviously, CEB slices are always equal or greater than their MEB counterparts.

The benchmarks selected for the experiments are the following:

- `ATM.csp`. This specification represents an Automated Teller Machine. The slicing criterion is `(Menu,getmoney)`, i.e., we are interested in determining what parts of the specification must be executed before the menu option `getmoney` is chosen in the ATM.

- `RobotControlling.csp`. This example describes a game in which four robots move in a maze. The slicing criterion is `(Referee,winner2)`, i.e., we want to know what parts of the system could be executed before the second robot becomes the winner.

- `Buses.csp`. This example describes a bus service with two buses running in parallel. The slicing criterion is `(BUS37, pay90)`, i.e., we are interested in determining what could and could not happen before the user payed at bus 37.

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

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

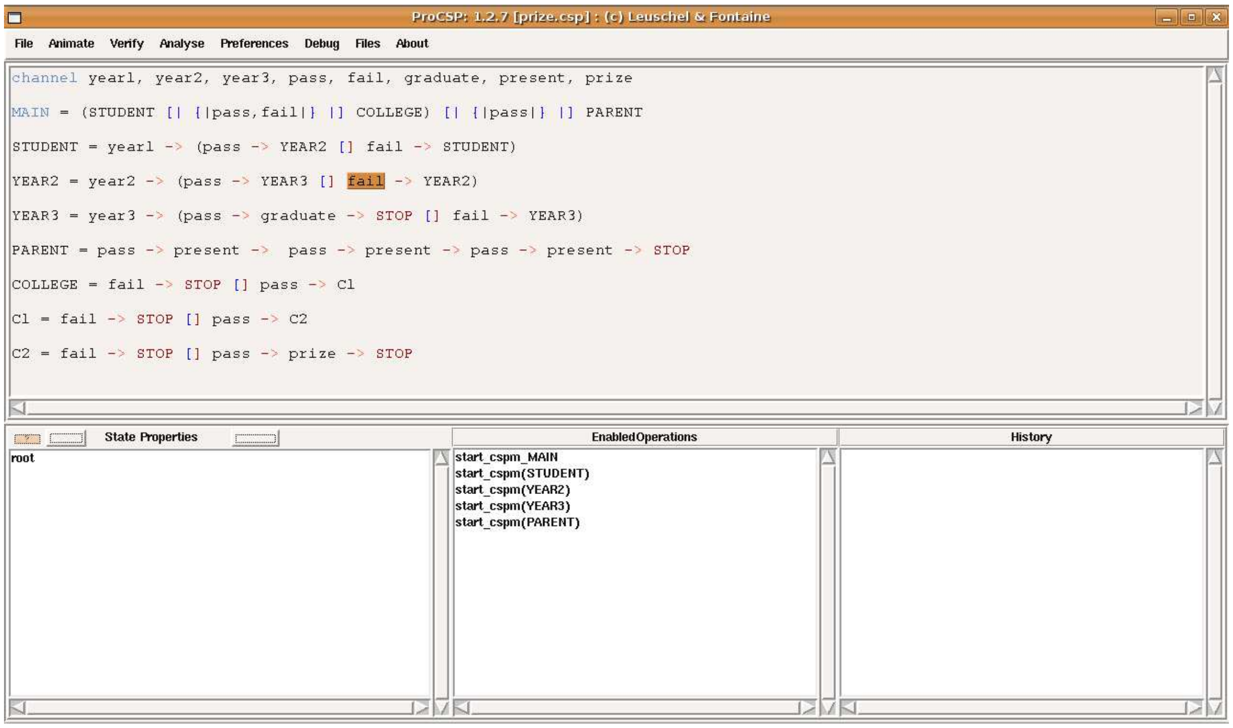
- `TrafficLights.csp`. This specification defines two cars driving in parallel on different streets. The first car can only circulate in two streets. The second car can only circulate in a third street. Each street has one traffic light for cars controlling. The slicing criterion is `(STREET3,park)`, i.e., we are interested in determining what parts of the specification must be executed before the second car parks on the third street.

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

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

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

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



```
channel year1, year2, year3, pass, fail, graduate, present, prize

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

STUDENT = year1 -> (pass -> YEAR2 [] fail -> STUDENT)

YEAR2 = year2 -> (pass -> YEAR3 [] fail -> YEAR2)

YEAR3 = year3 -> (pass -> graduate -> STOP [] fail -> YEAR3)

PARENT = pass -> present ->  pass -> present -> pass -> present -> STOP

COLLEGE = fail -> STOP [] pass -> C1

C1 = fail -> STOP [] pass -> C2

C2 = fail -> STOP [] pass -> prize -> STOP
```

```
root
```

```
start_cspm_MAIN
start_cspm(STUDENT)
start_cspm(YEAR2)
start_cspm(YEAR3)
start_cspm(PARENT)
```

Figure 7: Selecting the slicing criterion

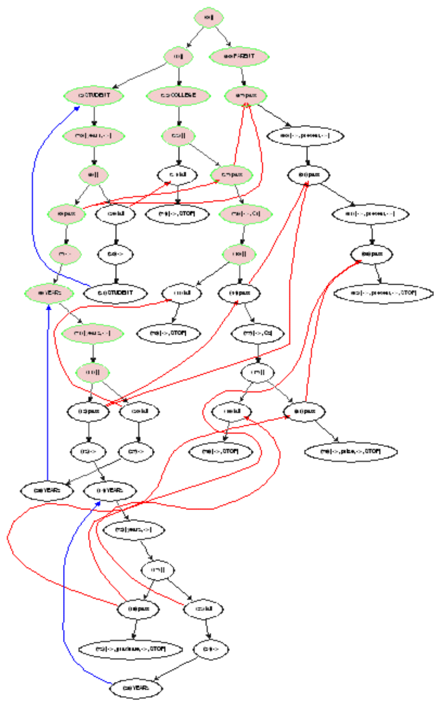


Figure 8: CSCFG of Example A.1