SOC: A Slicer for CSP Specifications *

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

Marisa Llorens Javier Oliver
Josep Silva Salvador Tamarit
Technical University of Valencia, Camino de Vera S/N
E-46022, Valencia, Spain
mllorens/fjoliver/jsilva/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

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 isolate the portion of code which contributed to 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 analyses based on program slicing have 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) a animator and model checker which can deal with CSP-specifications (2). 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 relies 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 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 daughter (indeed if she passed the first year) because this specification does not force the parent to give a present to his daughter until she has passed the second year. This is not so obvious from the specification, and* SOC *can help to understand the real meaning of the specification.*

* This work has been partially supported by the EU (FEDER and FP7 research project 214158 DEPLOY), by DAAD (PPP D/06/12830) and the Spanish MEC/MICINN under grants TIN2005-09207-C03-02, TIN2008-06622-C03-02, and *Acción Integrada* HA2006-0008; and by the Generalitat Valenciana under grant GVPRE/2008/001

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PEPM'09, January 19–20, 2009, Savannah, Georgia, USA.
Copyright © 2009 ACM 978-1-60558-327-3/09/01. . . $5.00

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

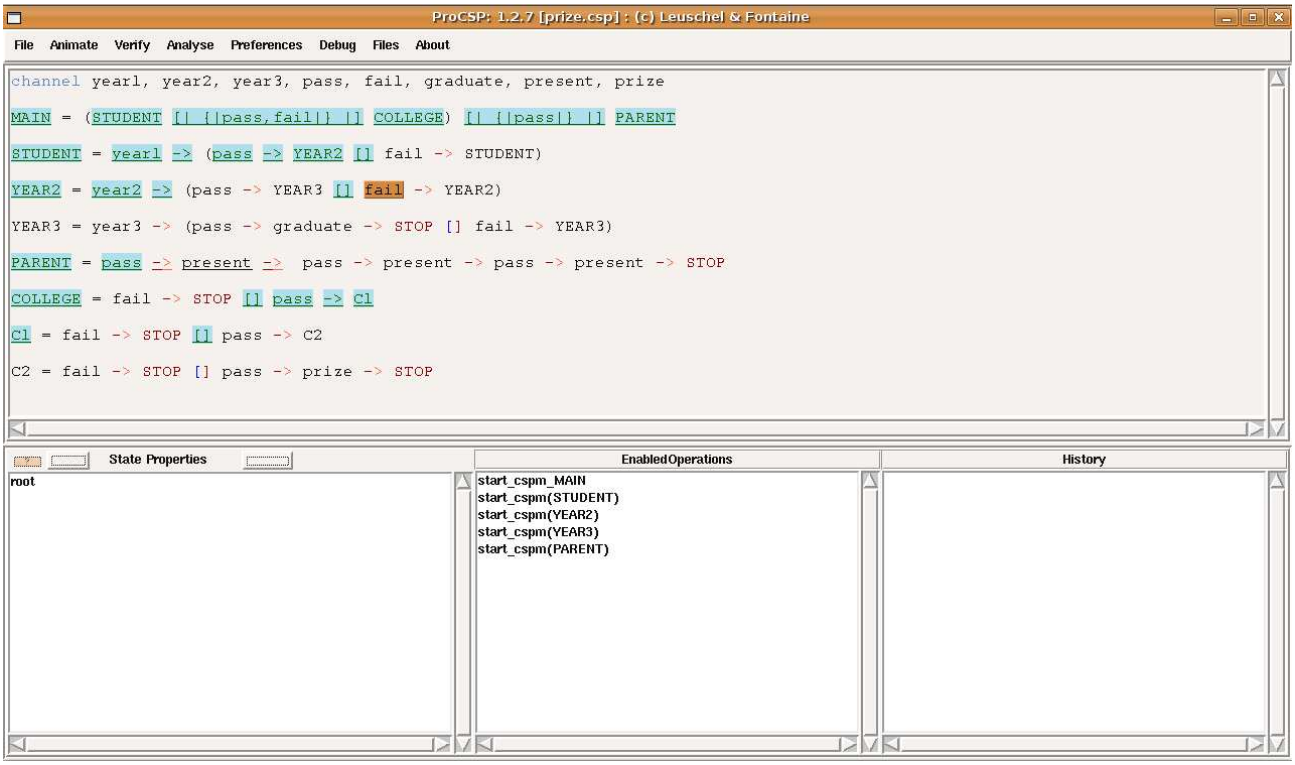

Figure 1. Slice of a CSP specification produced by *SOC*

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 (those not colored) by "`STOP`*" or by "*$\rightarrow$ `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 original 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

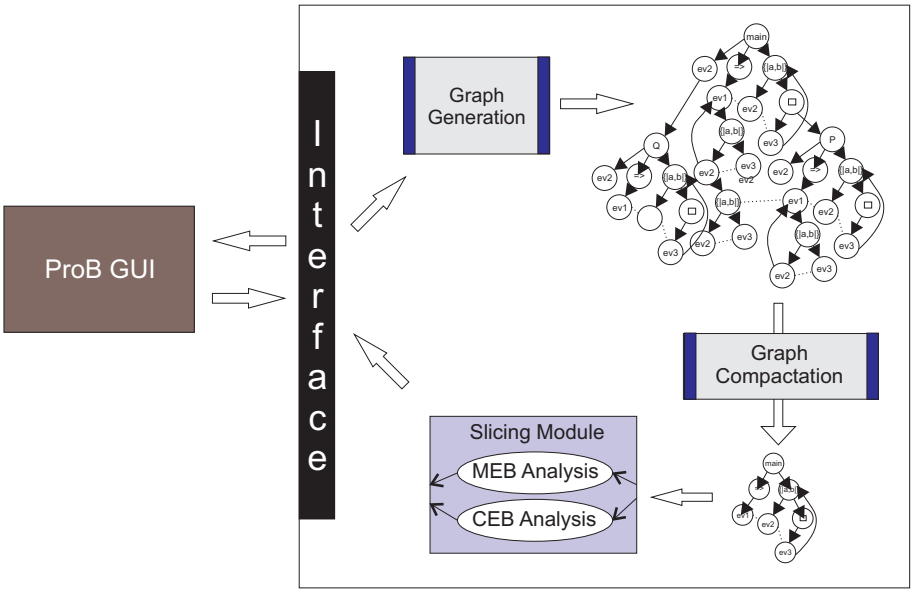

Figure 2. Slicer's Architecture

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 Compaction

The original definition of the CSCFG is not compacted enough 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 compaction not only reduces the size of the stored CSCFG, but it also speeds up the slicing process due to the reduced number of nodes to be processed.

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 compaction 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 the following benchmarks:

- `ATM.csp`. This specification represents an Automated Teller Machine. The slicing criterion is `(Menu,getmoney)`, i.e., we are interested in determining what parts of the specification must be executed before the menu option `getmoney` is chosen in the ATM.
- `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 wins.
- `Buses.csp`. This example (by Simon Gay) 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 2.1 (also by Simon Gay). The slicing criterion is `(YEAR2,fail)`, i.e., we are interested in determining what parts of the specification must be executed before the student fails in the second year.
- `Phils.csp`. This is a simple version of the dining philosophers problem. In this example, the slicing criterion is `(PHIL221, DropFork2)`, i.e., we want to know what happened before the second philosopher dropped the second fork.
- `TrafficLights.csp`. This specification defines two cars driving in parallel on different streets with traffic lights for cars controlling. The slicing criterion is `(STREET3,park)`, i.e., we are interested in 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

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 1. Benchmark time results

benchmark	Ori_CSCFG	Com_CSCFG	(%)	MEB Slice	CEB Slice
ATM.csp	163 nodes	110 nodes	67.48 %	48 nodes	61 nodes
RobotControlling.csp	339 nodes	123 nodes	36.28 %	36 nodes	109 nodes
Buses.csp	36 nodes	24 nodes	66.67 %	11 nodes	11 nodes
Prize.csp	70 nodes	49 nodes	70.00 %	17 nodes	18 nodes
Phils.csp	181 nodes	57 nodes	31.49 %	7 nodes	44 nodes
TrafficLights.csp	197 nodes	112 nodes	56.85 %	17 nodes	72 nodes
Processors.csp	30 nodes	15 nodes	50.00 %	8 nodes	9 nodes
ComplexSynchronization.csp	179 nodes	122 nodes	68.16 %	100 nodes	116 nodes
Computers.csp	53 nodes	34 nodes	64.15 %	28 nodes	28 nodes
Highways.csp	103 nodes	60 nodes	58.25 %	14 nodes	20 nodes

Table 2. Benchmark size results

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.

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

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

For each benchmark, Table 1 summarizes the time spent to generate the compacted CSCFG (this includes the generation plus the compaction phases), to produce the MEB and CEB slices (since CEB analysis uses MEB analysis, CEB's time corresponds only to the time spent after performing the MEB analysis), and the total time. Table 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. Column `(%)` shows the percentage of the compacted CSCFG' size with respect to the original CSCFG. Finally, columns `MEB Slice` and `CEB Slice` show respectively the size of the MEB and CEB CSCFG' slices. Clearly, CEB slices are always equal or greater than their MEB counterparts.

The CSCFG compaction technique 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. Table 2 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 wrote and adapted the ProB CSP parser which computes source code positions.

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 *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. *Communications ACM*, 26(1):100–106, 1983.

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

[6] M. Leuschel, M. Llorens, J. Oliver, J. Silva, and S. Tamarit. Static slicing of CSP specifications. In *Proceedings of 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.