Parallel Computing 53 (2016) 1–22

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Dynamic slicing of concurrent specification languages☆

M. Llorens[a,*], J. Oliver[a], J. Silva[a], S. Tamarit[b]

[a] Departamento de Sistemas Informáticos y Computación, Universitat Politècnica de València, Valencia, Spain
[b] Babel Research Group Fac. Informática, Universidad Politécnica de Madrid, Campus de Montegancedo, s/n. 28660 Boadilla del Monte, Spain

ARTICLE INFO

Article history:
Received 10 June 2015
Revised 29 October 2015
Accepted 10 January 2016
Available online 30 January 2016

Keywords:
Concurrent programming
CSP
Slicing

ABSTRACT

Dynamic slicing is a technique to extract the part of the program (called slice) that influences or is influenced, in a particular execution, by a given point of interest in the source code (called slicing criterion). Since a single execution is considered, the technique often uses a trace of this execution to analyze data and control dependencies. In this work we present the first formulation and implementation of dynamic slicing in the context of CSP. Most of the ideas presented can be directly applied to other concurrent specification languages such as Promela or CCS, but we center the discussion and the implementation on CSP. We base our technique on a new data structure to represent CSP computations called track. A track is a data structure which represents the sequence of expressions that have been evaluated during the computation, and moreover, it is labeled with the location of these expressions in the specification. The implementation of a dynamic slicer for CSP is useful for debugging, program comprehension, and program specialization, and it is also interesting from a theoretical perspective because CSP introduces difficulties such as heavy concurrency and non-determinism, synchronizations, frequent absence of data dependence, etc.

1. Introduction

Communicating Sequential Processes (CSP) [12,25] is one of the most widespread concurrent specification languages. The study and transformation of CSP specifications often uses different analyses such as deadlock analysis [17], reliability analysis [13], security analysis [23] and program slicing [32], which are based on a data structure able to represent computations through the use of traces [5].

However, standard CSP traces are not adequate for those analyses that need to relate the trace with the source code. One of these analyses is dynamic slicing [26,30], a technique to extract the part of a program (called dynamic slice) associated with a given slicing criterion. Concretely, a dynamic slice is the part of the program that influences or is influenced by a given point of interest in the source code for a given single execution of a program. In this work we claim that tracks [22] are an ideal data structure for dynamic slicing, and based on tracks, we present the first formulation and implementation of dynamic slicing in the context of CSP.

☆ This work has been partially supported by the EU (FEDER) and the Spanish *Ministerio de Economía y Competitividad* under Grant TIN2013-44742-C4-1-R and by the *Generalitat Valenciana* under Grant PROMETEOII/2015/013 (SmartLogic). Salvador Tamarit was partially supported by Madrid regional projects N-GREENS Software-CM (S2013/ICE-2731), and by European Union project POLCA (STREP FP7-ICT-2013.3.4 610686).

* Corresponding author. Tel.: +34 963877007.
E-mail addresses: mllorens@dsic.upv.es (M. Llorens), fjoliver@dsic.upv.es (J. Oliver), jsilva@dsic.upv.es (J. Silva), stamarit@babel.ls.fi.upm.es (S. Tamarit).

http://dx.doi.org/10.1016/j.parco.2016.01.006
0167-8191/

A CSP track is a data structure that represents the sequence of expressions that have been evaluated during one computation, labeled with the location of these expressions in the specification. In contrast, a (standard) CSP trace is the sequence of events that occur during the computation [25]. Therefore, a CSP track is much more informative than a CSP trace because the former not only contains a lot of information about original program structures but it also explicitly relates the sequence of events with the parts of the specification that caused these events.

Our implementation is the first dynamic program slicer for CSP specifications. It implements different versions of our technique that are useful for different goals such as debugging, program comprehension and program specialization. In all cases, the slicing process is completely automatic. The user only needs to load a CSP specification, specify a slicing criterion, and press a button. Then, the slicer automatically produces a computation and extracts the dynamic slice of this computation associated with the slicing criterion.

For the specification of the slicing criterion, we propose the use of a fresh channel (by default called `slice`) that can be located at any place(s) of the specification. Events of this channel do not interfere the execution of the specification and they are treated by the slicer as internal events (thus, they do not appear in the trace nor in the track). Allowing the use of more than one point is particularly interesting to face the problem of highly concurrent and non-deterministic processes. It is even possible to define the slicing criterion as a synchronized event, thus forcing the slicing criterion to happen in a specific synchronization. This is a novel idea that is introduced with our technique. To illustrate the slicing process and its internal data structure (the track) we use the following example.

Example 1. Consider the CPU diagram at the top of Fig. 1 used to simulate the process scheduler of a CPU. The CPU contains three main components: an Arithmetic Logic Unit (ALU), a Control Unit (CU), and a processes scheduler with a queue. The ALU is controlled by the CU. Only one process can access the CU each time. Therefore, the scheduler is in charge of granting access to the CU. For this, it uses a round robin strategy using the queue. In the figure, messages between components use solid arrows for query messages, and dashed arrows for answers.

This system can be modeled with the CSP specification[1] at the bottom of the figure (for the time being, the reader can ignore the difference between black and grey colors). The specification is buggy. It is syntactically correct, but the traces produced are not the expected ones. In particular, one can generate the following trace: ⟨`alu.3`, `working.3`, `cui.3`, `result.3`, `operation.3`⟩ that should be interpreted as: Process 3 requires access to the ALU, Process 3 continues working, Process 3 gets access to the CU, ALU produces a result for Process 3, CU asks ALU to solve an operation of Process 3.

Clearly, the two last events are in the wrong order. At this time we have a bug symptom, but we have to manually inspect the code to understand the problem. We are interested in determining what parts of the specification conducted the execution to produce event `result.3`, hence, we mark (it is marked with a box in the specification) expression `result.X` of process `Process(X)` as the slicing criterion. Our slicing technique automatically extracts the dynamic slice produced for that slicing criterion. It only contains the black code, which is enough to produce the error. Thus, it must contain a bug.

Looking again at the diagram in Fig. 1, we can see that the processes and the CPU communicate via two messages: `alu.proc` and `result.proc`. Therefore, it is clear that channels `alu` and `result` should be synchronized between the processes and the CPU. However, if we observe process `SYSTEM` in the slice, we can see that channel `result` has been accidentally replaced by `answer`. This is also obvious because process `Process` does not contain channel `answer`. Therefore, we can correct the error:

$$\texttt{SYSTEM} = \texttt{CPU} \underset{\{\texttt{alu,answer}\}}{\|} (\texttt{Process(1)} \,|||\, \texttt{Process(2)} \,|||\, \texttt{Process(3)})$$

should be

$$\texttt{SYSTEM} = \texttt{CPU} \underset{\{\texttt{alu,result}\}}{\|} (\texttt{Process(1)} \,|||\, \texttt{Process(2)} \,|||\, \texttt{Process(3)})$$

For the computation of slices we use tracks. For instance, a part of the track associated with the computation that produced the bug in Example 1 is depicted in Fig. 2. In the track, a directed graph, each node represents a term in the source code (it is easy to identify, e.g., the processes because their names are written in some nodes, and because the line and column of each expression is included in the node). Arcs are of two types: control-flow arcs (one-way solid arcs) and synchronizations (two-way dashed arcs). Control-flow arcs somehow represent a timeline. They represent the transition from one term to another term during the evaluation of the specification. Synchronizations always connect two events that happened at the same time. All this provides some interesting properties:

1. One can follow control-flow arcs to know the order in which source code terms where evaluated.
2. One node represents one specific term in one specific evaluation instant (control-flow arcs do not form loops).
3. A node with more than one synchronization arc represents a multiple synchronization (it should not be confused with a set of independent synchronizations at different moments), that is, all channels in a path of synchronization arcs must occur at the same time.

[1] We refer those readers non familiar with CSP syntax to Appendix A where we provide a brief introduction to CSP.