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

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 labelled 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.

Keywords: Concurrent Programming; CSP; Slicing.

[☆]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

Email 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)

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 [30, 26], 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.

A CSP track is a data structure that represents the sequence of expressions that have been evaluated during one computation, labelled 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 Figure 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 Figure 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:*

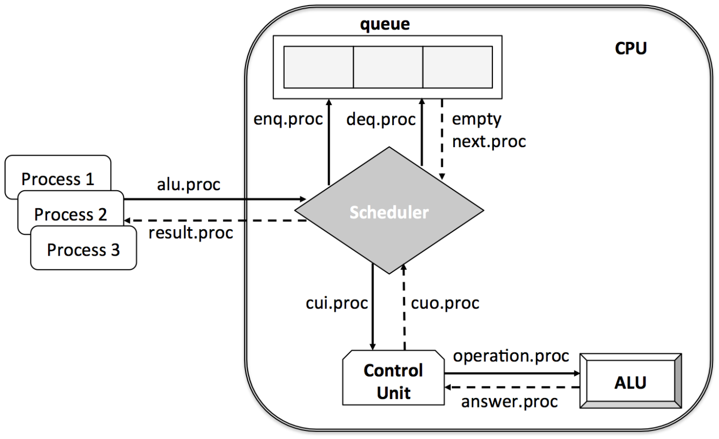
$$\texttt{SYSTEM = CPU} \underset{\{\texttt{alu,answer}\}}{||} \texttt{(Process(1) ||| Process(2) ||| Process(3))}$$

should be

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

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 Figure 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

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



```
channel operation, answer, cuo, cui, alu, working, result: {0..3}
channel shift, deq, empty
channel enq, next, left, right, comm: {0..3}

MAIN = SYSTEM
```

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

$$\texttt{Process(X) = alu!X} \to \texttt{working.X} \to \boxed{\texttt{result.X}} \to \texttt{SKIP}$$

$$\texttt{CPU = (Sched} \underset{\{\texttt{enq,deq,next,empty}\}}{||} \texttt{Queue)} \underset{\{\texttt{cui,cuo}\}}{||} \texttt{(CU} \underset{\{\texttt{operation,answer}\}}{||} \texttt{ALU)}$$

$$\texttt{Sched = Sched\_idle}$$

$$\texttt{Sched\_idle = alu?proc} \to \texttt{cui!proc} \to \texttt{Sched\_busy}$$

$$\begin{aligned}\texttt{Sched\_busy = }&\texttt{cuo?proc} \to \texttt{(result.proc} \to \texttt{Sched\_check)}\\ &\square\ \texttt{alu?proc} \to \texttt{enq!proc} \to \texttt{Sched\_busy}\end{aligned}$$

$$\begin{aligned}\texttt{Sched\_check = deq} \to \texttt{(}&\texttt{empty} \to \texttt{Sched\_idle}\\ &\square\ \texttt{next?proc} \to \texttt{cui!proc} \to \texttt{Sched\_busy)}\end{aligned}$$

$$\texttt{CU = cui?proc} \to \texttt{operation.proc} \to \texttt{answer.proc} \to \texttt{cuo!proc} \to \texttt{CU}$$

$$\texttt{ALU = operation?proc} \to \texttt{answer!proc} \to \texttt{ALU}$$

$$\texttt{Queue = (DQ(0)} \underset{\{\texttt{left,right,shift}\}}{||} \texttt{BUFF)}\backslash\{\texttt{left,right,shift}\}$$

$$\texttt{DQ(2) = deq} \to \texttt{shift} \to \texttt{X(2)}$$

$$\begin{aligned}\texttt{DQ(i) = }&\texttt{enq?x} \to \texttt{(left!x} \to \texttt{shift} \to \texttt{DQ(i+1))}\\ &\square\ \texttt{deq} \to \texttt{(empty} \to \texttt{DQ(0)} \not\prec \texttt{i==0} \not\succ \texttt{X(i))}\end{aligned}$$

$$\texttt{X(i) = right?y} \to \texttt{(next!y} \to \texttt{DQ(i-1))}\ \square\ \texttt{shift} \to \texttt{X(i)}$$

$$\texttt{BUFF = (CELL } \llbracket \texttt{right} \Re \texttt{comm} \rrbracket \underset{\{\texttt{comm}\}}{||} \texttt{CELL } \llbracket \texttt{left} \Re \texttt{comm} \rrbracket\texttt{)}\backslash\{\texttt{comm}\}$$

$$\texttt{CELL = left?x} \to \texttt{shift} \to \texttt{right!x} \to \texttt{CELL}$$

Figure 1: CPU diagram and its specification in CSP

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.

Thanks to these properties, a slice can be computed in linear time with a single traversal of the track. In particular, a backward dynamic slice can be computed traversing the track backwards from the slicing criterion. Hence, in the figure, those nodes colored in grey are the slice obtained selecting `result.X` as the slicing criterion (the node with the bold line).

Example 1 illustrates how dynamic slicing can be used for debugging, significantly reducing the amount of code that must be inspected to find a bug. Besides debugging, dynamic slicing could be used for program comprehension. Different slices show the parts of the specification really involved in one particular computation. Thus, they can help to understand the actual meaning of the specification.

Another interesting application is program specialization. Note that the slice produced is not executable, but it could be made executable by replacing the removed parts by `STOP`, or by $\rightarrow$ `STOP` if the removed expression has a prefix. For instance, process `SYSTEM` could be redefined as:

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

Hence, this transformation allows us to extract executable 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).

The rest of the paper has been organized as follows. First, in Section 2 we present and discuss the related work. In Section 3 we introduce some definitions and notation. Next, in Section 4, we present the formulation of dynamic slicing via tracking in the context of CSP. We describe our implementation and its empirical evaluation in Section 5. Finally, Section 6 concludes.

2. Related Work

Since it was originally defined by Weiser [32], program slicing has been applied to different formalisms that are not strictly programming languages, like

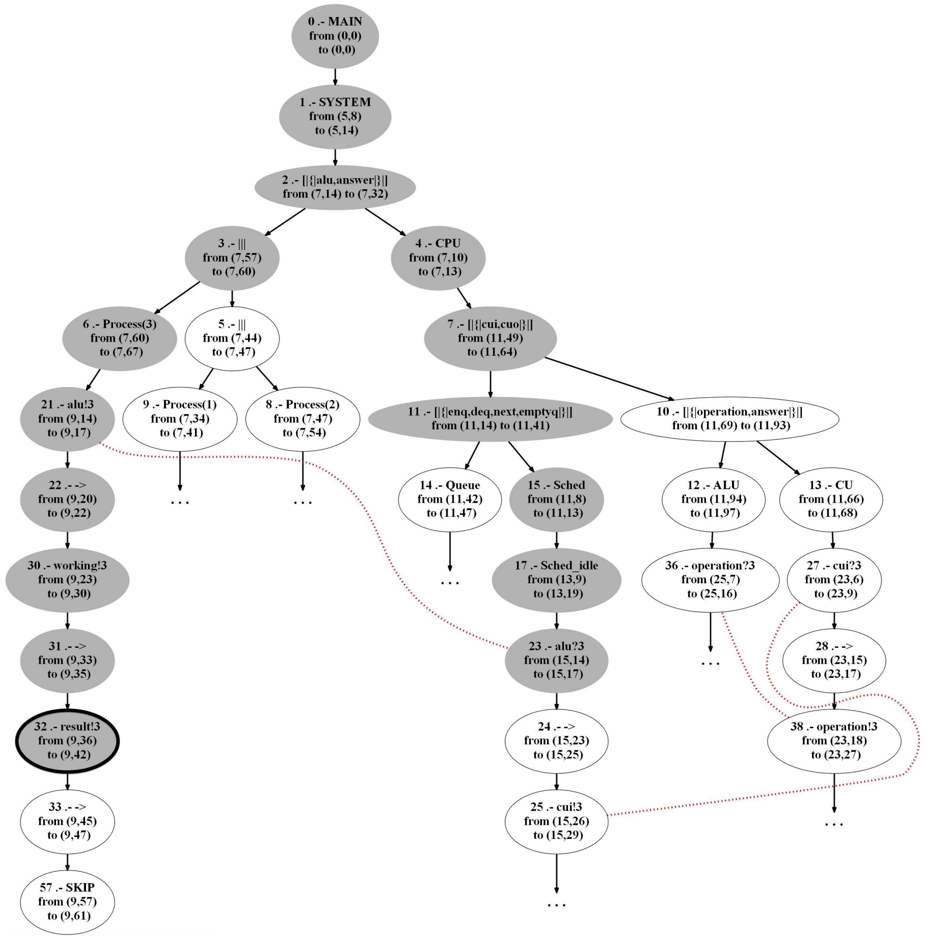

0 .- MAIN from (0,0) to (0,0)
1 .- SYSTEM from (5,8) to (5,14)
2 .- [|{|alu,answer|}|] from (7,14) to (7,32)
3 .- ||| from (7,57) to (7,60)
4 .- CPU from (7,10) to (7,13)
6 .- Process(3) from (7,60) to (7,67)
5 .- ||| from (7,44) to (7,47)
7 .- [|{|cui,cuo|}|] from (11,49) to (11,64)
21 .- alu!3 from (9,14) to (9,17)
9 .- Process(1) from (7,34) to (7,41)
8 .- Process(2) from (7,47) to (7,54)
11 .- [|{|enq,deq,next,emptyq|}|] from (11,14) to (11,41)
10 .- [|{|operation,answer|}|] from (11,69) to (11,93)
22 .- -> from (9,20) to (9,22)
14 .- Queue from (11,42) to (11,47)
15 .- Sched from (11,8) to (11,13)
12 .- ALU from (11,94) to (11,97)
13 .- CU from (11,66) to (11,68)
30 .- working!3 from (9,23) to (9,30)
17 .- Sched_idle from (13,9) to (13,19)
36 .- operation?3 from (25,7) to (25,16)
27 .- cui?3 from (23,6) to (23,9)
31 .- -> from (9,33) to (9,35)
23 .- alu?3 from (15,14) to (15,17)
28 .- -> from (23,15) to (23,17)
32 .- result!3 from (9,36) to (9,42)
24 .- -> from (15,23) to (15,25)
38 .- operation!3 from (23,18) to (23,27)
33 .- -> from (9,45) to (9,47)
25 .- cui!3 from (15,26) to (15,29)
57 .- SKIP from (9,57) to (9,61)

Figure 2: Partial track of the CPU specification

attribute grammars [27], hierarchical state machines [11], Petri nets [14], etc., and different slicing techniques have been specifically defined for concurrent programs (e.g., for analyzing concurrency bugs [31], or for race detection [29]).

Unfortunately, very little work has been carried out on slicing concurrent specification languages, and it is practically missing in CSP. Some notable exceptions are [4, 3, 21] that were proposed in the context of the *static* analysis of CSP. In the area of static slicing, analyses are often based on the use of a data structure that approximates all possible derivations of a CSP specification. Of course, this is radically different to our approach because a track is a dynamic structure that could be infinite, while their data structures are finite representations of possibly infinite derivations. Moreover, tracks are confident data structures, while their data structures are approximations. The similitude with our work appears in the fact that some of their data structures also use a mapping to the source code, and thus they are able to relate a derivation with the part of the source code that is needed to perform this derivation. For instance, Brückner and Wehrheim [4] proposed a data structure based on the standard *program dependence graph* [8] to slice CSP-OZ specifications or CSP-OZ-DC specifications [3], ignoring CSP synchronization and taking a LTL formulae constructed with OZ's variables as slicing criterion. It is useful for program slicing but it is insufficient for other analyses that need a *context-sensitive graph* [16] (i.e., each different process call has a different representation). Later, [21] proposed the *Context-sensitive Synchronized Control Flow Graph* (CSCFG) (a similar data structure that was context-sensitive and takes into account CSP synchronization) and two static analyses based on this graph that can be applied to CSP.

Dynamic slicing [15] however is missing in CSP. To the best of our knowledge, this work defines the first adaptation of dynamic program slicing for CSP. Of course, there exist different dynamic analyses and tools able to simulate specifications and analyze, e.g., deadlocks (often based on the *stable failures model*), and also the circumstances under which processes can livelock (often based on the *divergences model*) [25]. Two of the most important analysis tools for CSP are ProB [18, 19] and FDR3 [10]. Both of them allow the user to dynamically generate and explore the possible traces of a specification. Probably, the data structure produced by FDR3's debugger is the most similar to our notion of tracks. As it happens with our tracks, the debug viewer can represent several behaviors of particular machines all together, and how they relate (each behavior is represented with a sequence of events, and events that are vertically aligned are synchronised). But our tracks explicitly represent the relation between the sequence of events, and the source code that produced these events, in such a way that the analyst knows exactly what part of the code is being activated when an event, a choice, a parallel execution, etc. happens. Moreover, graphs produced by FDR3 are labelled transition systems (LTS), thus nodes represent states, and arcs model transitions through the occurrence of events. Contrarily, tracks are not LTS. In a track, nodes are not states, but terms of the source code (such as prefixes, choice operators or parallel operators).

There exist tools to convert CSP programs into other traditional languages such as C++ or Java. And these languages have available dynamic slicers. Thus, one may ask wether it is possible (or useful) to translate a CSP specification to an equivalent C program, slice the C program, and translate the sliced C program to CSP. The short answer is no, because the current technology to do this is not mature enough.

If we think about C++, the long answer is that *CSP++*[2] is a software synthesis tool for making specifications written in the machine-readable dialect of CSP (CSP_M[3]) executable via C++. Once the code has been transformed to C++, we could use a C++ dynamic program slicer. However, to the best of

[2]`http://www.uoguelph.ca/~gardnerw/csp++/`
[3]`http://www.cs.ox.ac.uk/ucs/cspm.pdf`

our knowledge, most of existing slicing tools for C/C++ perform static slicing (such as *Frama-C*[4]—only for the C language—or *CodeSurfer*[5]). One of the few publicly available dynamic slicing tools for C/C++ is *Giri*[6] that implements dynamic backwards slicing in *LLVM*[7] compiler. Giri supports three ways of specifying the slicing criterion: the return instruction at the main function, source code line number and LLVM instruction number. For a given input, it reports the dynamic slice of the program execution from the slice starting point in terms of LLVM instructions as well as the source code line numbers. Disappointingly, once we have produced the C++ slice, there does not exist a translation to go back to CSP. Thus, the slice should be interpreted at C++ level.

We have used this software and, unfortunately, the transformed C++ code is very (very!) different from the original CSP specification, and hard to understand, even if you know C++. In particular, it is hard to even identifying the original processes, choices, etc. in the transformed code. Only an expert in C++ familiarized with the transformation can interpret the dynamic slice with respect to the original CSP specification. Moreover, a slice formed by C++ line numbers is much less intuitive than the graphical slices produced by CSP-Tracker. In addition, CSP-Tracker can return an executable slice, Giri no.

If we focus on Java, among tools to convert CSP to Java, we can find *JCircus*[8] a tool that automatically translates Circus programs into Java. It is based on a translation strategy that uses the *JCSP*[9] library to implement some of the CSP constructs of Circus[10] (a formal language that combines the Z and CSP notations). The operational semantics of Circus presents some important differences with respect to CSP_M.

If we were able to translate CSP to Java, we could then use the *JSlice*[11] and *JavaSlicer*[12] dynamic slicing tools for Java programs. In his PhD thesis, Hammacher[13] compared JSlice and JavaSlicer, and he concluded that JavaSlicer is superior. In any case, neither of them would be better than CSP-Tracker. On the one hand, JSlice is not maintained since 2008 —it only works for versions of JDK previous to 1.4—. Moreover, JSlice does not produces a graphical view of the slice. On the other hand, JavaSlicer is able to produce graphical slices but, unfortunately, this only works for very (very!) small graphs, because of the poor layouting algorithms shipped with JUNG. Even if we produced a slice of

[4]`http://frama-c.com/`

[5]`http://www.grammatech.com/research/technologies/codesurfer`

[6]`https://github.com/liuml07/giri`

[7]`http://llvm.org/`

[8]`https://www.cs.york.ac.uk/circus/tools/jcsp.php`

[9]`http://www.cs.kent.ac.uk/projects/ofa/jcsp/`

[10]`https://www.cs.york.ac.uk/circus/`

[11]`http://jslice.sourceforge.net/`

[12]`https://www.st.cs.uni-saarland.de/javaslicer/`
`https://github.com/hammacher/javaslicer`

[13]`https://www.st.cs.uni-saarland.de/publications/files/hammacher-bachthesis-2008.pdf`

the transformed Java code, currently, there is no tool to convert a Java program to a CSP specification. Thus, slices should be interpreted at the level of Java.

3. Preliminary definitions and notation

In this section we introduce some notation, and provide definitions used in the rest of the paper. We start with a mechanism to decompose CSP specifications.

Traditionally, in imperative languages, program slicing works at the level of code lines. This means that program slices are formed by a subset of the lines in the original code. This is clearly inappropriate for CSP where code is not organized in lines, but in expressions. Hence, we want to reduce the granularity level of slices to the level of expressions. Indeed, we want to maximally reduce the granularity level thus being able to identify any single literal in a CSP specification.

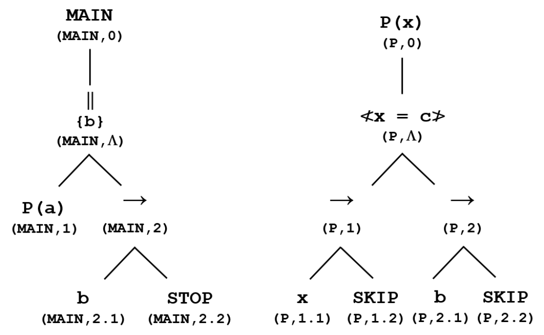
To uniquely identify each literal in a CSP specification we use labels (that we call *specification positions*), which roughly correspond to nodes in the CSP specification's abstract syntax tree. A *specification position* [22] is a pair (N, w) where N is the name of a CSP process and w is a chain of natural numbers separated by dots (we use Λ to denote the empty chain).

Example 2. *The following CSP specification has been labelled with specification positions (they are underlined):*

$$\texttt{MAIN}_{\underline{(\texttt{MAIN},0)}} \;=\; \texttt{P(a)}_{\underline{(\texttt{MAIN},1)}} \underset{\{\texttt{b}\}}{\|}_{\underline{(\texttt{MAIN},\Lambda)}} (\texttt{b}_{\underline{(\texttt{MAIN},2.1)}} \rightarrow_{\underline{(\texttt{MAIN},2)}} \texttt{STOP}_{\underline{(\texttt{MAIN},2.2)}})$$

$$\texttt{P(x)}_{\underline{(\texttt{P},0)}} \;=\; (\texttt{x}_{\underline{(\texttt{P},1.1)}} \rightarrow_{\underline{(\texttt{P},1)}} \texttt{SKIP}_{\underline{(\texttt{P},1.2)}}) \not< \texttt{x = c} \not>_{\underline{(\texttt{P},\Lambda)}} (\texttt{b}_{\underline{(\texttt{P},2.1)}} \rightarrow_{\underline{(\texttt{P},2)}} \texttt{SKIP}_{\underline{(\texttt{P},2.2)}})$$

All terms are uniquely labelled because labels keep the order of the associated abstract syntax tree:



We also need to define the notions of *rewriting step* and *derivation*. For this, we need to base our definition on some CSP semantics. Our implementation is based on the standard CSP semantics (see, e.g., [25]), but here, in the theoretical development, the semantics is left open. In CSP, event-based semantics evolve processes due to the occurrence of events of either an alphabet Σ, the internal event τ, or the success event $\checkmark$, which denotes the successful termination of a process, often indicated with Ω. We allow the use of any particular semantics provided that it is formed of rules $\dfrac{\Theta}{P \xrightarrow{e} P'}$ where P and P' are processes, Θ is a possibly empty set of rewriting steps, and e is an event. An example of such a semantics can be found in, e.g., [21] and in Appendix A.

Definition 1 (Rewriting Step, Derivation, Subderivation). *Given a CSP process P, a* rewriting step *for P, denoted by $P \overset{\Theta}{\rightsquigarrow} P'$, is the transformation of P into P' by using a rule of the CSP semantics. Therefore, $P \overset{\Theta}{\rightsquigarrow} P'$ iff a rule of the form $\dfrac{\Theta}{P \xrightarrow{e} P'}$ is applicable, where $e \in \Sigma \cup \{\tau, \checkmark\}$ and Θ is a (possibly empty) set of rewriting steps. Given a CSP process P_0, we say that the sequence $\mathcal{D} = P_0 \overset{\Theta_0}{\rightsquigarrow} \dots \overset{\Theta_n}{\rightsquigarrow} P_{n+1}$, $n \geq 0$, is a* derivation *of P_0 iff $\forall\, i, 0 \leq i \leq n, P_i \overset{\Theta_i}{\rightsquigarrow} P_{i+1}$ is a rewriting step. A subderivation of $\mathcal{D}$ is any $\mathcal{D}' = P_i \overset{\Theta_i}{\rightsquigarrow} \dots \overset{\Theta_j}{\rightsquigarrow} P_{j+1}$, $0 \leq i < j \leq n$. We say that the derivation is* complete *iff there is no possible rewriting step for P_{n+1}. We say that the derivation has* successfully finished *iff P_{n+1} is Ω.*

Example 3. *One (possible) complete derivation of Example 2 is the derivation of Figure 3 where the rules applied in each rewriting step are labelled with their names (with subderivations as subindexes) in the standard CSP semantics:* (Process Call)*,* (Synchronized Parallelism 1)*,* (Synchronized Parallelism 3)*,* (Conditional Choice 2) *and* (Prefixing) *(abbrev.* (PC)*,* (SP1)*,* (SP3)*,* (CC2) *and* (Pref)*, respectively). For instance, $E_1 \underset{(\mathsf{SP1}\ _{\mathsf{PC}})}{\rightsquigarrow} E_2$ means that expression E_1 is transformed into E_2 with a rewriting step performed with the standard semantics rule* (Synchronized Parallelism 1) *that in turn performs a subderivation using the standard semantics rule* (Process Call)*. All terms that trigger the rules of the semantics are labelled with their specification positions (in grey). Observe that this derivation has not successfully finished.*

$$\begin{array}{lcl}
\mathtt{MAIN}_{(\mathtt{MAIN},0)} & \underset{(\mathsf{PC})}{\rightsquigarrow} & \mathtt{P(a)}_{(\mathtt{MAIN},1)} \underset{\{\mathtt{b}\}}{\parallel}_{(\mathtt{MAIN},\Lambda)} (\mathtt{b} \to \mathtt{STOP}) \\
 & \underset{(\mathsf{SP1}\ _{\mathsf{PC}})}{\rightsquigarrow} & ((\mathtt{a} \to \mathtt{SKIP}) \not< \mathtt{a} = \mathtt{c} \not>_{(\mathtt{P},\Lambda)} (\mathtt{b} \to \mathtt{SKIP})) \underset{\{\mathtt{b}\}}{\parallel} (\mathtt{b} \to \mathtt{STOP}) \\
 & \underset{(\mathsf{SP1}\ _{\mathsf{CC2}})}{\rightsquigarrow} & (\mathtt{b}_{(\mathtt{P},2.1)} \to_{(\mathtt{P},2)} \mathtt{SKIP}) \underset{\{\mathtt{b}\}}{\parallel} (\mathtt{b}_{(\mathtt{MAIN},2.1)} \to_{(\mathtt{MAIN},2)} \mathtt{STOP}) \\
 & \underset{(\mathsf{SP3}\ _{\mathsf{Pref}})}{\rightsquigarrow} & \mathtt{SKIP}_{(\mathtt{P},2.2)} \underset{\{\mathtt{b}\}}{\parallel} \mathtt{STOP} \\
 & \underset{(\mathsf{SP1}\ _{\mathsf{SKIP}})}{\rightsquigarrow} & \Omega \underset{\{\mathtt{b}\}}{\parallel} \mathtt{STOP}
\end{array}$$

Figure 3: Derivation associated with the specification of Example 2

A *trajectory* of a CSP derivation is a sequence formed by the specification positions of the expressions that trigger the rules of the semantics (in the evaluation order) in the derivation. To establish a concrete evaluation order in the derivation we consider that, in a rule $\dfrac{\Theta}{P \xrightarrow{e} P'}$, the term evaluated in this step is P, and the associated specification position is the root of the tree formed by term P. For instance, if $P = a \rightarrow SKIP \mid\mid Q(3)$, whose root is $\mid\mid$, then the specification position evaluated is the one associated to $\mid\mid$. There is only one exception: if P is a prefixing of the form $a_\alpha \rightarrow_\beta R$, whose root is $\rightarrow$, then we consider that α is evaluated first, and then β.

If Θ contains one single rewriting step, then P is evaluated first, and then the term evaluated in Θ. If Θ contains more than one rewriting step (e.g., in the standard CSP operational semantics, only two rewriting steps are possible, and this means that two processes are being synchronized), then the terms evaluated in Θ are evaluated at the same time. Finally, in $P \overset{\Theta}{\rightsquigarrow} P' \overset{\Theta'}{\rightsquigarrow} P''$, P is evaluated before P', and the terms evaluated in Θ are evaluated before the terms evaluated in Θ'.

Definition 2 (Trajectory of a CSP derivation). *Given a CSP derivation $\mathcal{D} = P_0 \overset{\Theta_0}{\rightsquigarrow} \ldots P_n \overset{\Theta_n}{\rightsquigarrow} P_{n+1}$, $n \geq 0$, a* trajectory *of $\mathcal{D}$ for a given input x is the sequence of sets of specification positions $\langle\{\alpha\}, ..., \{..., \gamma\}\rangle$ associated to the terms evaluated in each rewriting step of the derivation $\mathcal{D}$ in the evaluation order, where $\alpha = (P_0, 0)$, and γ is the specification position of the last term evaluated in process P_n.*

For instance, the trajectory of the derivation in Figure 3 is: $\langle\{(\texttt{MAIN}, 0)\}, \{(\texttt{MAIN}, \Lambda)\}, \{(\texttt{MAIN}, 1)\}, \{(\texttt{MAIN}, \Lambda)\}, \{(\texttt{P}, \Lambda)\}, \{(\texttt{MAIN}, \Lambda)\}, \{(\texttt{P}, 2.1), (\texttt{MAIN}, 2.1)\}, \{(\texttt{P}, 2)\}, \{(\texttt{MAIN}, 2)\}, \{(\texttt{MAIN}, \Lambda)\}, \{(\texttt{P}, 2.2)\}\rangle$.

4. Dynamic Slicing in CSP

Traditionally, dynamic slicing has been based on the concept of *trajectory* [15]. Given an execution of an imperative program, the trajectory of this execution is a list with the sequence of instructions executed (in the execution order).

Definition 3 (Dynamic slicing criterion [15]). *Let T be a trajectory of program P on input x. A* slicing criterion *of program P executed on input x is a triple $C = (x, I^q, V)$, where I is an instruction at position q on T and V is a subset of variables in P.*

Later, the meaning of I^q was changed by "*the instruction I when it is executed the qth time*" [1, 2]. In the following, we will use this second meaning for being the most extended.

Clearly, this definition is inappropriate for CSP for two fundamental reasons. First, CSP is not an imperative language, and thus it is not formed with instructions. Hence, I does not make sense in CSP. Moreover, the use of variables is not always necessary (many CSP specifications do not use variables), thus V is not a good choice to determine what we are interested in.

Hence, we need to provide a new definition of slicing criterion suitable for CSP. Basically, the problem is that I and V were designed to select a point in an imperative program. In CSP, rather than dividing the code in instructions, it is divided into processes which in turn are divided into expressions that can be further decomposed in terms. Therefore, a point in a CSP specification should refer to an expression or a term. Thus, we use specification positions as a mechanism to identify expressions.

On the other hand, what determines a CSP computation are events. In fact, there exist several models to study CSP computations and their expressivity, and they all rely on events and traces of events [12, 25]. Hence, instead of using variables (V), in CSP it seems more adequate to use events as the relevant execution information. Thus, a slicing criterion should specify a particular event of interest that should be identified as a term in some process definition.

We can provide now a first definition of dynamic slicing criterion for CSP.

Definition 4 (CSP dynamic slicing criterion (version 1)). *Let $\mathcal{S}$ be a CSP specification, and $\mathcal{T}$ the trajectory of a derivation of $\mathcal{S}$ for a given input x. A* slicing criterion *for $\mathcal{T}$ is a pair $\mathcal{C} = (x, s^q)$, where s is a specification position of $\mathcal{S}$, and $q > 0$ is an integer denoting the qth occurrence of s in $\mathcal{T}$.*

Note that this definition defines one point in the trajectory. Hence, this definition is valid for both forwards (collecting those specification positions that may be affected by that point) and backwards (collecting those specification positions that may affect that point) slicing. In the following, we will use the term dynamic slicing to implicitly refer to backwards dynamic slicing.

We could base our definition of dynamic slice in CSP on the standard notion of dynamic slice (i.e., *a subset of the original program formed by all those instructions that do affect the slicing criterion in all possible trajectories induced by x*). However, due to the combinatorial explosion of trajectories in CSP (even if no input is provided, due to parallelism) we can be more precise, and only consider one derivation when computing slices. Roughly speaking, a dynamic slice in CSP is formed by the set of specification positions that affect the slicing criterion in a given derivation. Of course, this is very vague, because we have no idea of what "affect" means, since the standard notion of data dependence and control dependence is not applicable to CSP.

Therefore, to properly define dynamic slices, we need to define first what "affect" means. Intuitively, given a concrete derivation, a term t_1 does influence another term t_2, if the later cannot be computed when the former is replaced by `STOP`. This means that t_1 is needed to compute t_2 in that derivation.

For the definition of influence, we introduce the notion of `STOP`*-substitution* and `STOP`*-subderivation*. Roughly, a `STOP`-substitution of a CSP process is the same process where a subprocess has been replaced by `STOP`. It is a way of preventing this subprocess to be executed.

Definition 5 (`STOP`-substitution). *Let P be a CSP process, and let α be an arbitrary specification position in P. A `STOP`-substitution of P at α, denoted with $Subs(P, \alpha)$, is process P where the term at α has been substituted by `STOP`.*

Example 4. *Consider the following* `STOP`*-substitution:*

$$Subs(\ \underline{\mathtt{P}_{\mathtt{(MAIN,1)}}}\ \underline{\underset{\{\mathtt{b}\}}{\|}\ _{\mathtt{(MAIN,\Lambda)}}}\underline{\mathtt{Q}_{\mathtt{(MAIN,2)}}}\ ,\mathtt{(MAIN,1)})\ =\ \underline{\mathtt{STOP}_{\mathtt{(MAIN,1)}}}\ \underline{\underset{\{\mathtt{b}\}}{\|}\ _{\mathtt{(MAIN,\Lambda)}}}\underline{\mathtt{Q}_{\mathtt{(MAIN,2)}}}$$

A `STOP`-subderivation of a given derivation $\mathcal{D}$ is a derivation that performs a subsequence of the rewriting steps in $\mathcal{D}$, in the same order, but where some terms have been replaced by `STOP`. This concept is very useful to define CSP slices, because it allows us to know whether a term is needed in a derivation to reach a slicing criterion (another term). Clearly, if a term t is replaced by `STOP`, and the derivation can still reach the slicing criterion, then we can say that t does not influence the computation of the slicing criterion in that particular derivation.

Definition 6 (`STOP`-subderivation). *Given two derivations:*
$\mathcal{D} = P_0 \overset{\Theta_0}{\rightsquigarrow} \ldots \overset{\Theta_n}{\rightsquigarrow} P_{n+1},\ n \geq 0$
$\mathcal{D}' = P_0 \overset{\Theta_0}{\rightsquigarrow} \ldots \overset{\Theta_{i-1}}{\rightsquigarrow} P_i \overset{\Theta_i}{\rightsquigarrow} P'_{i+1} \ldots \overset{\Theta_j}{\rightsquigarrow} P'_{j+1},\ 0 \leq i < j \leq n$
$\mathcal{D}'$ *is a* `STOP`*-subderivation of* $\mathcal{D}$ *if and only if:*

- $P_0 \overset{\Theta_0}{\rightsquigarrow} \ldots \overset{\Theta_{i-1}}{\rightsquigarrow} P_i$ *is a subderivation of* $\mathcal{D}$*,*
- P'_{i+1} *is a* `STOP`*-substitution of* P_{i+1} *at some* α*, and*
- *the trajectory of* $P_i \overset{\Theta_i}{\rightsquigarrow} P'_{i+1} \ldots \overset{\Theta_j}{\rightsquigarrow} P'_{j+1}$ *is a subsequence of the trajectory of* $P_i \overset{\Theta_0}{\rightsquigarrow} \ldots \overset{\Theta_n}{\rightsquigarrow} P_{n+1}$.

It is important to highlight that the notion of subsequence used in the previous definition is the standard mathematical notion (i.e., a subsequence is a sequence that can be derived from another sequence by deleting some elements without changing the order of the remaining elements). This is important because this ensures that $\mathcal{D}$ and $\mathcal{D}'$ follow the same control path. Concretely, $\mathcal{D}'$ performs the same rewriting steps than $\mathcal{D}$ except for those that cannot be done because the evaluated term has been replaced by `STOP`.

With a `STOP`-subderivation we can formally define a notion of influence between terms in a derivation.

Definition 7 (Influence in a Derivation). *Let $\mathcal{T}$ be the trajectory of a derivation $\mathcal{D} = P_0 \overset{\Theta_0}{\rightsquigarrow} \ldots \overset{\Theta_n}{\rightsquigarrow} P_{n+1}$, $n \geq 0$. Let α and β specification positions of terms in P_i and P_j, $0 \leq i < j \leq n+1$, respectively. We say that α does* influence *β ($\alpha \Rightarrow \beta$), in $\mathcal{D}$ if and only if there exists a* `STOP`*-subderivation of $\mathcal{D}$:*
$\mathcal{D}' = P_0 \overset{\Theta_0}{\rightsquigarrow} \ldots \overset{\Theta_{i-1}}{\rightsquigarrow} P'_i \overset{\Theta_i}{\rightsquigarrow} \ldots \overset{\Theta_{j-1}}{\rightsquigarrow} P'_j,\ 0 \leq i < j \leq n+1,$
and the following holds:

- P'_i *is the* `STOP`*-substitution* $Subs(P_i, \alpha)$*, and*
- β *does not exist in* P'_j*, or* β *in* P'_j *is* `STOP`.

Based on the definition of `STOP`-subderivation and influence, we provide a formal definition of dynamic slice for CSP.

Definition 8 (Dynamic Slice). *Let $\mathcal{S}$ be a CSP specification, let $\mathcal{C} = (x, s^q)$ be a* slicing criterion *for $\mathcal{S}$, and $\mathcal{T}$ the trajectory of a derivation of $\mathcal{S}$ for a given input x. A* dynamic slice *associated with $\mathcal{C}$ is formed by the set of specification positions: $\mathcal{S}lice = \{\alpha \in \mathcal{T} \mid \alpha \Rightarrow^* t\}$, where t is the qth occurrence of s in $\mathcal{T}$.*

4.1. Dynamic slicing of CSP based on tracks

In all paradigms, the efficient computation of dynamic slices has been done using a data structure that represents an execution such as the Dynamic Dependence Graph [1] (used in imperative languages) or the Redex Trails [24] (used in functional languages). Neither of these data structures are appropriate for CSP. We claim that tracks are an ideal data structure to compute dynamic slices in CSP.

Definition 9 (Track). *Given a CSP specification $\mathcal{S}$, and a trajectory $\mathcal{T}$ of a derivation $\mathcal{D}$ of $\mathcal{S}$ for a given input x, the* track *of $\mathcal{T}$ is a directed acyclic graph $\mathcal{G} = (N, E_c, E_s)$, where N is a set of nodes uniquely identified with a natural number and labelled with specification positions (e.g., a_α where a is the node and α is a specification position), and arcs are divided into two groups:*

- control-flow arcs *(E_c) are a set of one-way arcs (denoted with $\mapsto$) representing the control-flow between two nodes, and*
- synchronization arcs *(E_s) are a set of two-way arcs (denoted with $\leftrightarrow$) representing the synchronization of two (event) nodes;*

and

1. *E_c contains a control-flow arc $a_\alpha \mapsto a'_\beta$ iff there exists a dynamic control dependence between $\alpha \in \mathcal{T}$ and $\beta \in \mathcal{T}$, and*
2. *E_s contains a synchronization arc $a \leftrightarrow a'$ for each synchronization occurring in $\mathcal{D}$ where a and a' are the nodes of the synchronized events.*

The only nodes in N are the nodes induced by E_c and E_s.

For the purpose of this work, how tracks are constructed is not relevant. The interested reader is referred to [22] where there is a formalization of tracks (including the formal definition of dynamic control dependence) and their construction. The important property of tracks that is relevant for dynamic slicing is that there is a one-to-one relationship between a given computation of a CSP specification and a track. Hence, tracks uniquely represent computations because they just represent a trajectory with explicit information about synchronizations. The track associated with the trajectory of the derivation of Figure 3 is shown in Figure 4 (for the time being ignore the different shapes and colors of nodes).

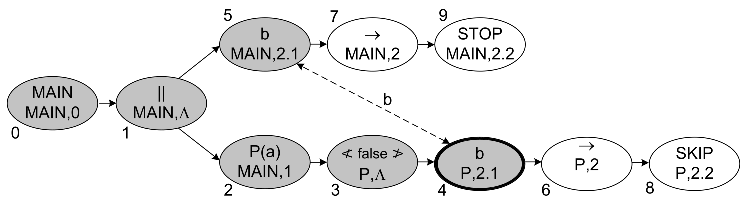


Figure 4: Track associated with the trajectory of the derivation in Figure 3

Lemma 1 (Slicing criterion in a track). *Let $\mathcal{S}$ be a CSP specification, let $\mathcal{C} = (x, s^q)$ be a* slicing criterion *for $\mathcal{S}$, and let $\mathcal{G} = (N, E_c, E_s)$ be the* track *associated with a derivation of $\mathcal{S}$ for the input x. s^q uniquely identifies a single node in N or a set of synchronized nodes in N.*

Proof. It follows from the fact that (i) the track contains a node with the specification position of each element in the trajectory of the derivation, and (ii) nodes keep the order of the trajectory through control arcs. Hence, according to the slicing criterion, there must exist at least q nodes with specification position s, and

- there exists only one node with specification position s in position q, or
- there exist more than one node with specification position s in position q, but they must be synchronized, thus, they all happen at the same time, and hence, in the same position (q). ■

Lemma 1 relates the slicing criterion with a track and, moreover, it states that a slicing criterion is really a single node of the track, or a set of nodes connected by synchronization arcs. Thus, the slicing criterion can be specified by selecting just one node. For instance, given the CSP specification of Example 2 and a slicing criterion $\mathcal{C} = (\{\}, (\mathtt{P}, 2.1)^1)$, in the track of Figure 4, $\mathcal{C}$ is the node identified with number 4 (the node with the bold line).

Definition 10 (Track Slice). *Let $\mathcal{S}$ be a CSP specification, let $\mathcal{C} = (x, s^q)$ be a* slicing criterion *for $\mathcal{S}$, and let $\mathcal{G} = (N, E)$ be the* track *associated with a derivation of S for the input x. The* backward track slice *associated with $\mathcal{C}$ and $\mathcal{G}$ is formed by the set of specification positions of those nodes in N reachable from a backward traversal of $\mathcal{G}$ from the node in N identified by s^q.*

As an example, in the track of Figure 4, those nodes colored in grey are the track slice obtained selecting the node with the bold line as the slicing criterion.

We can ensure that a track slice is a dynamic slice for the following reasons:

- There is a one-to-one relation between track and trajectory,
- $(a_\alpha \mapsto a'_\beta)$ implies $\alpha \Rightarrow \beta$, and
- $(a_\alpha \leftrightarrow a'_\beta)$ implies $\alpha \Rightarrow \beta$ and $\beta \Rightarrow \alpha$.

Therefore, track slices are a means to compute dynamic slices with a cost linear with respect to the size of the track.

4.2. A generalized dynamic slicing criterion

We can generalize the definition of dynamic slicing criterion using a set of specification positions instead of a single one.

Definition 11 (CSP dynamic slicing criterion (version 2)). *Let $\mathcal{S}$ be a CSP specification, and $\mathcal{T}$ the trajectory of a derivation of $\mathcal{S}$ for a given input x. A* slicing criterion *for $\mathcal{T}$ is a pair $\mathcal{C} = (x, \{s\}^q)$, where $\{s\}$ is a set of specification positions of $\mathcal{S}$, and $q > 0$ is an integer denoting the qth occurrence in $\mathcal{T}$ of the specification positions in $\{s\}$.*

Using this generalized definition of slicing criterion slices can be obtained in the same way.

Example 5. *Consider again the CSP specification in Example 2. If we are interested in the part of the specification executed before one of the two branches of the parallelism operator is executed for the first time, we can define the dynamic slicing criterion $\mathcal{C} = (x, \{(\mathtt{MAIN}, 1), (\mathtt{MAIN}, 2)\}^1)$. Observe that (due to the restriction imposed by the synchronization) one branch must be always executed before the other. Therefore, this criterion would produce the slice shown in the track of Figure 5 (it is formed by the set of specification positions of those nodes colored in grey).*

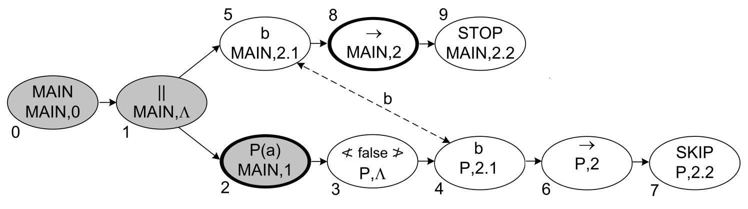


Figure 5: Slice of Example 5

Definition 11 is general enough as to identify any set of expressions in the specification (not only channels, because the set of specification positions can refer to any literal, e.g., process names, parallel operators, choice operators, etc.). This is very convenient for debugging, and for program comprehension. However, when analyzing a CSP specification in practice, analysts are usually interested in the occurrence of events, regardless of where in the specification they appear.

Therefore, we provide another definition of slicing criterion more oriented to the implementation. It is a particular case of Definition 11 where a set of channels is used instead of a set of specification positions.

Definition 12 (CSP dynamic slicing criterion (version 3)). *Let $\mathcal{S}$ be a CSP specification, and $\mathcal{T}$ the trajectory of a derivation of $\mathcal{S}$ for a given input x. A* slicing criterion *for $\mathcal{T}$ is a pair $\mathcal{C} = (x, \{a\}^q)$, where $\{a\} \subseteq \Sigma$ (a subset of the specification channels), and $q > 0$ is an integer denoting the qth occurrence of channels of $\{a\}$ in $\mathcal{T}$.*

A slicing criterion with Definition 12 for the CSP specification in Example 5 is, e.g., $\mathcal{C} = (x, \{\mathtt{b}\}^1)$. Obviously, this is equivalent to the slicing criterion $\mathcal{C} = (x, \{(\mathtt{MAIN}, 2.1), (\mathtt{P}, 2.1)\}^1)$, and the associated slice is the same (the one shown in Figure 4).

4.3. Correctness and completeness

In program slicing, the general objective is to produce techniques that are complete. Correctness has been proven undecidable in the general case (it would imply to compute the minimal slice) [32]. Hence, the slices produced with our technique are complete but not necessarily correct. Concretely, given a slicing criterion $\mathcal{C} = (x, \{a\}^q)$, the slice produced always contains all the specification positions needed to produce a trace for the input x such that the trace contains at least q elements of $\{a\}$. But it is also possible that the slice contains specification positions that are not needed to produce that trace.

Theorem 1 (Slice completeness). *Let $\mathcal{S}$ be a CSP specification, and let $\mathcal{C} = (x, \{a\}^q)$ be a* slicing criterion *for $\mathcal{S}$. The slice associated with $\mathcal{C}$ contains all specification positions needed to produce a trace t for the input x such that $t \in \Sigma^*$, $\{a\} \subseteq \Sigma$, and $t \downarrow \{a\} = q$.*

Proof idea. Let $\mathcal{G} = (N, E)$ be the *track* associated with a derivation of S for the input x. First, it is trivial to extend Lemma 1 to consider a set of events instead of a single one. Therefore, by Lemma 1 we know that $\mathcal{C}$ refers to either one single node in the track, or to a set of synchronized nodes. In both cases, the slice associated with $\mathcal{C}$ collects all nodes traversing backwards control dependence arcs and synchronization arcs from the nodes associated with the slicing criterion. By Def. 12, t contains at least q elements of $\{a\}$, i.e. $t \downarrow \{a\} = q$. Moreover, according to Theorem 3 (Track correctness) in [22], $\mathcal{G}$ contains all events in trace t keeping their order with control dependence arcs. Therefore, the specification positions in the slice can reproduce the same trace t for the input x until the events in $\{a\}$ appear q times. ∎

Theorem 2 (Slice incorrectness). *Let $\mathcal{S}$ be a CSP specification, and let $\mathcal{C} = (x, \{a\}^q)$ be a* slicing criterion *for $\mathcal{S}$. The slice associated with $\mathcal{C}$ can contain specification positions not needed to produce a trace t for the input x such that $t \in \Sigma^*$, $\{a\} \subseteq \Sigma$, and $t \downarrow \{a\} = q$.*

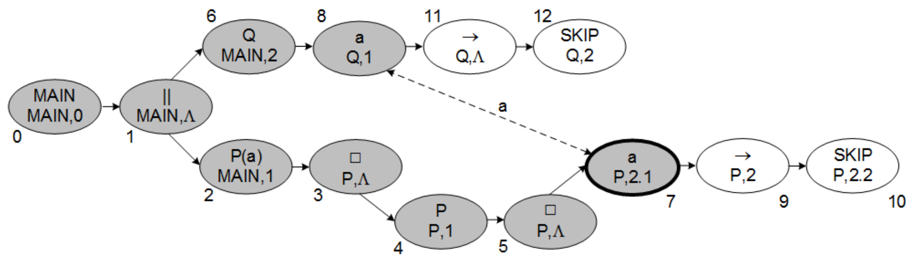
Proof. We proof this theorem with a counterexample showing that a slice can collect unnecessary specification positions. Consider the following CSP specification:

$$\mathtt{MAIN}_{\underline{(\mathtt{MAIN},0)}} = \mathtt{P}_{\underline{(\mathtt{MAIN},1)}} \underset{\{\mathtt{a}\}}{||}_{\underline{(\mathtt{MAIN},\Lambda)}} \mathtt{Q}_{\underline{(\mathtt{MAIN},2)}}$$

$$\mathtt{P}_{\underline{(\mathtt{P},0)}} = \mathtt{P}_{\underline{(\mathtt{P},1)}} \;\square_{\underline{(\mathtt{P},\Lambda)}}\; (\mathtt{a}_{\underline{(\mathtt{P},2.1)}} \rightarrow_{\underline{(\mathtt{P},2)}} \mathtt{SKIP}_{\underline{(\mathtt{P},2.2)}})$$

$$\mathtt{Q}_{\underline{(\mathtt{Q},0)}} = \mathtt{a}_{\underline{(\mathtt{Q},1)}} \rightarrow_{\underline{(\mathtt{Q},\Lambda)}} \mathtt{SKIP}_{\underline{(\mathtt{Q},2)}}$$

A track associated with this specification and the slice computed for the slicing criterion $\mathcal{C} = (\{\}, \{(\mathtt{P}, 2.1)\}^1)$ are shown bellow.



Clearly, this slice is not correct, because it includes specification position $(\mathtt{P}, 1)$, which is not needed to produce the trace $\langle \mathtt{a} \rangle$ (the left branch of the choice is a loop that do not contribute to the trace, but it is included in the slice). ∎

5. Implementation and Empirical Evaluation

We developed a tool called CSP-Tracker that implements a CSP interpreter with a tracker and a slicer. The interpreter executes a CSP specification and simultaneously produces the track associated with the performed derivation. Then, the user can specify a slicing criterion and the slice is automatically computed.

For the specification of the slicing criterion in the CSP specification, a fresh channel (called `slice`) can be placed at any place(s) of the specification. It is also allowed the use of more than one slicing point (i.e, placing `slice` at different locations), which is very convenient in presence of highly concurrent and non-deterministic processes. It is even possible to specify the slicing criterion as a synchronized event, thus forcing the slicing criterion to happen in a specific synchronization.

Example 6. *In the following CSP specification, the programmer has specified a synchronization as the slicing criterion (those events inside the boxes):*

$$\mathtt{MAIN} = \mathtt{P} \underset{\{\boxed{\mathtt{slice}}\}}{||} \mathtt{Q}$$

$$\mathtt{P} = \mathtt{a} \rightarrow \mathtt{b} \rightarrow \boxed{\mathtt{slice}} \rightarrow \mathtt{SKIP}$$

$$\mathtt{Q} = \mathtt{a} \rightarrow \mathtt{b} \rightarrow \mathtt{c} \rightarrow \boxed{\mathtt{slice}} \rightarrow \mathtt{SKIP}$$

Because `slice` *is part of the synchronization set in the parallelism operator, it can only happen as a synchronization between* `P` *and* `Q`*. The slice will contain all parts of the specification needed to produce this synchronization.*

For the purpose of slicing, tracks are an internal data structure totally transparent for the user. However, tracks can be also a useful tool to graphically inspect the computation before or after specifying a slicing criterion. Hence, the tracker incorporates mechanisms to produce colored graphs that represent the tracks in an intuitive way. In CSP-Tracker, both the tracking and slicing processes are completely automatic. Once the user has loaded a CSP specification, he can (automatically) produce a derivation and the tool internally generates the associated track. Then, the tool asks for the number of occurrence of `slice` he is interested in. This information is enough to generate the slice. Both the track and the trace, and also the slice, can be stored in a file, or displayed in the screen by generating *Graphviz*[14] graphs.

CSP-Tracker is publicly available including its source code as a GitHub repository:

`https://github.com/mistupv/csp_tracker/`

There is also a web interface useful to test the tool. It can be found at:

`http://kaz.dsic.upv.es/csp_tracker`

Figure 6 shows a screenshot of the interface of the tool showing the executable slice of the specification in Example 1.

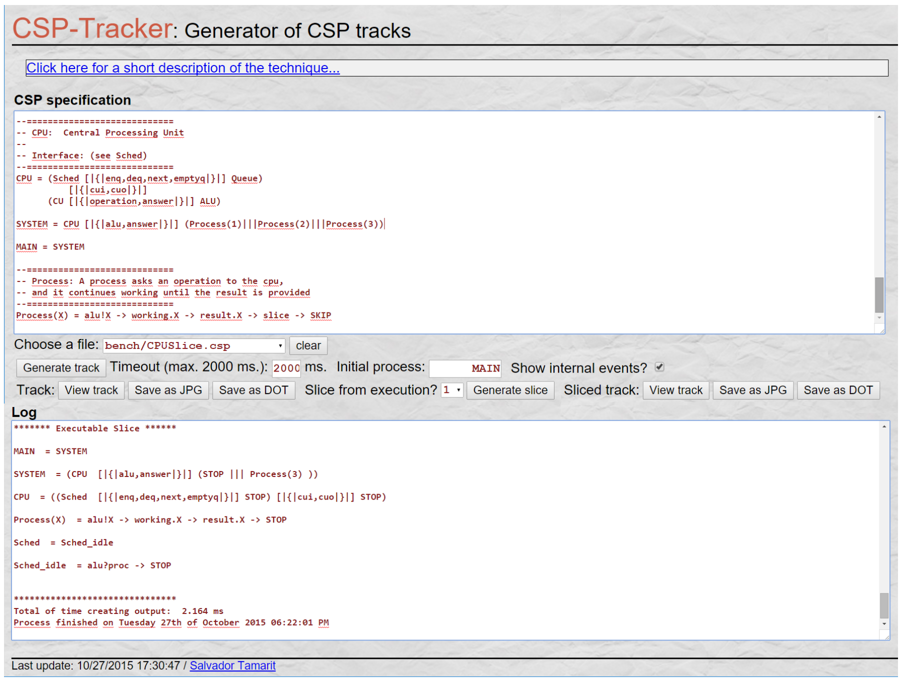
5.1. Architecture of CSP-Tracker

The information collected by CSP-Tracker is *dynamic*, and thus the analyses performed are very precise. However, it is prepared to also integrate static analyses. CSP-Tracker has been implemented in Erlang[15]. The election of Erlang was very conscious because Erlang is one of the most efficient languages for the use of multiple processes and concurrent programming [33, 7]; and it provides concurrent capabilities that enhance the execution of CSP specifications with the use of efficient message passing. In particular, with Erlang we can use truly concurrent processes to implement interleaving and synchronized parallelism.

Figure 7 summarizes the internal architecture of CSP-Tracker. In the figure, the dark rectangles represent modules that are described in the following:

- `ProB's CSP parser`: The CSP parser is part of ProB [18, 19], which is one of the most extended IDE for CSP. It translates a CSP specification into a Prolog representation. This Prolog structure acts as an intermediate language that is prepared to perform other analyses for CSP. For instance,

[14]`http://www.graphviz.org/`
[15]`http://www.erlang.org/`


CSP-Tracker: Generator of CSP tracks
Click here for a short description of the technique...
CSP specification

```
--============================
-- CPU:  Central Processing Unit
--
-- Interface: (see Sched)
--============================
CPU = (Sched [|{|enq,deq,next,emptyq|}|] Queue)
          [|{|cui,cuo|}|]
      (CU [|{|operation,answer|}|] ALU)

SYSTEM = CPU [|{|alu,answer|}|] (Process(1)|||Process(2)|||Process(3))

MAIN = SYSTEM

--============================
-- Process: A process asks an operation to the cpu,
-- and it continues working until the result is provided
--============================
Process(X) = alu!X -> working.X -> result.X -> slice -> SKIP
```

Choose a file: bench/CPUSlice.csp
Timeout (max. 2000 ms.): 2000 ms. Initial process: MAIN Show internal events? ✓
Slice from execution? 1

clear
Generate track
Track: View track Save as JPG Save as DOT
Generate slice
Sliced track: View track Save as JPG Save as DOT
Log

```
******* Executable Slice ******

MAIN  = SYSTEM

SYSTEM  = (CPU  [|{|alu,answer|}|] (STOP ||| Process(3) ))

CPU  = ((Sched  [|{|enq,deq,next,emptyq|}|] STOP) [|{|cui,cuo|}|] STOP)

Process(X)  = alu!X -> working.X -> result.X -> STOP

Sched  = Sched_idle

Sched_idle  = alu?proc -> STOP


*******************************
Total of time creating output:  2.164 ms
Process finished on Tuesday 27th of October 2015 06:22:01 PM
```

Last update: 10/27/2015 17:30:47 / Salvador Tamarit

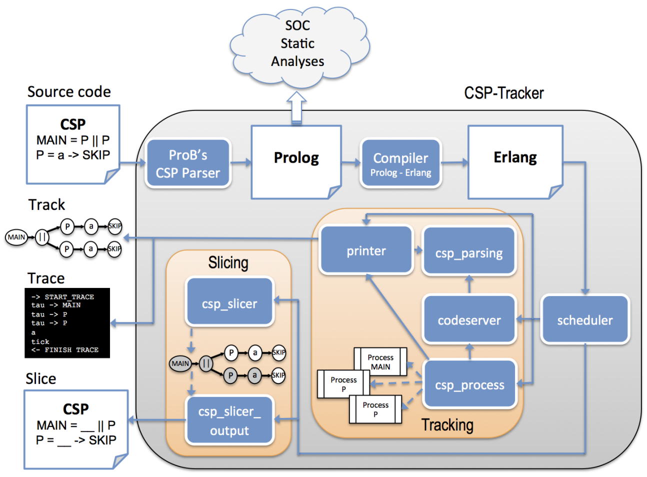
Figure 6: Screenshot of the web interface of CSP-Tracker

the tool SOC [20] uses this Prolog representation to perform different *static* analyses.

This module is in charge of assigning specification positions. While in the theoretical framework (for the sake of simplicity) we use natural numbers to represent specification positions, in the implementation we use lines and columns to identify literals which is much more convenient and useful for the programmer. This can be observed in Figure 2. For instance, node 4 with literal `CPU` has the specification position `from (7,10) to (7,13)`, which means that `CPU` appears in the source code between columns 10 and 13 of line 7.

- `Compiler Prolog-Erlang`: It uses a Prolog module to produces an Erlang representation equivalent to the Prolog structure. This step does not imply any semantic transformation. This is just a change in the syntactic representation that is almost straightforward because the syntax of Erlang was initially based on Prolog, so there are many similarities between them.

- `scheduler`: This module initializes and coordinates the other modules. First, it loads the Erlang code produced and then it creates all the Erlang processes needed by the tool. Finally, it starts the execution of the initial CSP process (by default `MAIN`) to generate the track. Once the track has



```
-> START_TRACE
tau -> MAIN
tau -> P
tau -> P
a
tick
<- FINISH TRACE
```

Figure 7: CSP-Tracker's Architecture

been generated, it traverses the track from the slicing criterion to extract the slice.

- `codeserver`: This module specifies a process that runs uninterruptedly during the generation of the track. It behaves as a server that stores all the information about the code of the CSP processes. It waits for requests and serves them. A request is in fact a message that contains a process call. Then, `codeserver` returns a message containing the right hand side of the called process with the parameters substituted by the actual values of the arguments in the call.

- `printer`: This module also specifies a process that runs uninterruptedly and acts as a server. In this case, the requests contain information that should be used to print the trace of the execution or to generate the part of the track that represents the ongoing execution. To graphically show tracks, we use Graphviz.

- `csp_process`: This module creates one Erlang process for each CSP process in the specification. All created processes run in parallel and synchronize via message passing when needed. Each of these processes interacts with `codeserver` and `printer` to perform process calls and generate the graph when required. For instance, the execution of a prefixing `a` $\rightarrow$ `P` calls `printer` to print `a` in the shell and add the corresponding nodes and arcs to the track. Then, it calls `codeserver` to get the right hand side of `P`, and create a new process that represents this right hand side.

- `csp_parsing`: This module is basically a library with common functionality for the other modules.

- `csp_slicer`: This module is in charge of mapping the slicing criterion to the track (i.e., identifying the corresponding node), and then traversing the track backwards from the slicing criterion to collect the slice. How the programmer specifies the slicing criterion was one of the problems we faced. Given a CSP code, specifying the slicing criterion (i.e., one specific term, and a number) can be awkward for the programmer whether it is specified with specification positions or with line and column. We finally found a very easy solution that avoids the problem of manually identifying terms in the specification. The programmer can specify a slicing criterion by just placing a fresh channel (called `slice`) in the point of interest. Then, the tool asks for the desired occurrence of events of this channel. This is simple, and very powerful and expressive, because it allows the programmer to specify more than one point of interest, thus potentially focussing on any number of events of interest.

  Example 7. *In the following CSP specification, the programmer has specified two different points for the slicing criterion (those inside the boxes):*

  $\texttt{channel request, printout, netAccess, slice}$

  $$\texttt{MAIN = Server} \underset{\{\texttt{request,printout}\}}{\parallel} \texttt{(Printer1} \mid\mid\mid \texttt{Printer2)}$$

  $$\texttt{Printer1 = request?X} \rightarrow \texttt{printout.X} \rightarrow \boxed{\texttt{slice}} \rightarrow \texttt{Printer1}$$

  $$\texttt{Printer2 = netAccess} \rightarrow \texttt{request?X} \rightarrow \texttt{printout.X} \rightarrow \boxed{\texttt{slice}} \rightarrow \texttt{Printer2}$$

  ```
  Server = request!42 -> printout.42 -> Server
  ```

  *This means that we are interested in all parts of the specification needed to print a document (no matter what printer does it). The slicer would automatically ask "*What occurrence are you interested in?*". If we answer 1, then only one of the printers is needed. If we answer, e.g., 5, then, depending on the trace, both printers could participate in the slice. This flexibility of the slicing criterion that can be located at any place, and that is composed of one or more points of interest has the expressivity of simultaneous slicing [6]. Because* `slice` *can be placed at any point, we can compute slices for several (not necessarily the same) events, just before the execution of a specific process or synchronization, etc.*

- `csp_slicer_output`: This module extracts a slice from the nodes collected in the track. The slice is composed of all specification positions of the nodes collected. This is useful for debugging, but useless, e.g., for program specialization, because these specification positions alone would produce a syntactically incorrect specification. Therefore, this module also replaces the gaps in the specification by `STOP`. Hence, two outputs are possible: (i) a non-executable part of the specification useful for debugging (see Figure 1), or also (ii) a well-formed CSP specification able to produce the same computation until the slicing criterion is reached (see Figure 6).

5.2. Empirical evaluation

We conducted several experiments to empirically evaluate CSP-Tracker. These experiments provide a precise and quantitative idea of the performance of the execution, track generation, and slice computation.

For the evaluation, we selected a set of heterogeneous benchmarks from public CSP repositories. All of them have been previously used to test other CSP tools and techniques. The source code of the benchmarks can be found at:

`https://github.com/mistupv/csp_tracker/tree/master/benchmarks/`

In order to evaluate the performance of our tool, we strictly followed the methodology proposed in [9, 28]. All benchmarks were executed in the same hardware configuration: Intel® Xeon® Processor E5504 (4 cores, 4M Cache, 2.00 GHz) with 16GB RAM. During the execution of the benchmarks all processes of the system except CSP-Tracker were stopped to avoid interference of external programs. Each benchmark was repeatedly executed 1001 times with a timeout of 2 seconds. To ensure real independence, the first iteration was always discarded (to avoid influence of dynamically loaded libraries persisting in physical memory, data persisting in the disk cache, etc.). Thus, we obtained 1000 statistical values. Then, we computed the 0.99 confidence interval across the computed values from the different 1000 executions.

This process was repeated for each of the 10 benchmarks (10,000 executions in total), and it produced the set of measures shown in Table 1. We computed both the arithmetic and the harmonic mean to study the effect of statistical dispersion, which was sufficiently low as to use the arithmetic mean in our table results.

The threshold of 2 seconds was selected, both in the experiments and in the web interface, because it is enough to produce long tracks composed of more than 1500 nodes. The good scalability of the tool permits to generate long Erlang computations composed of many parallel process in 2 seconds. In the web interface the threshold is needed for security reasons, so that our server cannot be attacked with demands for infinite or very long computations. In the experiments, the threshold is useful to compare the track produced by different specifications (with different levels of complexity, number of parallel processes, number of synchronizations, etc.) executed exactly the same time.

In the tables, we use the notation $[_a\ b\ _c]$ that represents a symmetric 0.99 confidence interval between a and c with center in b. Times are measured in milliseconds and memory sizes are represented in bytes. Each column in the tables is described in the following:

- `Benchmark` is the name of the benchmark.
- To measure time performance we have:

Benchmark	CSP2Erlang (ms)	Runtime (ms)	Generate Slice (ms)
ABP.csp	[480.03 499.71 519.39]	[807.13 1054.07 1301.00]	[17.01 23.21 29.41]
ATM.csp	[313.60 314.86 316.12]	[298.41 357.09 415.77]	[14.31 15.75 17.18]
Buses.csp	[124.91 125.61 126.30]	[1.10 1.11 1.11]	[0.85 0.86 0.87]
CPU.csp	[188.35 189.29 190.23]	[10.16 10.23 10.31]	[1.82 1.83 1.84]
Disk.csp	[192.10 192.94 193.78]	[23.61 23.86 24.12]	[5.50 5.56 5.63]
Loop.csp	[128.53 129.20 129.87]	[2006.23 2006.53 2006.82]	[18.47 18.61 18.75]
Oven.csp	[197.75 198.70 199.65]	[35.13 39.43 43.73]	[3.49 3.60 3.71]
ProdCons.csp	[137.01 137.84 138.68]	[2005.67 2005.94 2006.21]	[19.46 19.57 19.68]
ReadWrite.csp	[143.11 144.05 144.99]	[2004.57 2004.93 2005.28]	[21.95 22.66 23.38]
Traffic.csp	[161.36 162.02 162.69]	[6.02 6.47 6.92]	[0.19 0.24 0.29]
Average	[206.68 209.22 212.17]	[719.80 750.97 782.13]	[10.31 11.19 12.07]

(a) Execution time results

Benchmark	#Nodes	#Control Edges	#Sync. Edges
ABP.csp	[101.07 135.85 170.63]	[79.15 105.12 131.09]	[8.84 12.67 16.50]
ATM.csp	[353.60 393.97 434.34]	[235.65 261.36 287.08]	[55.88 63.14 70.39]
Buses.csp	[24.00 24.00 24.00]	[16.00 16.00 16.00]	[3.00 3.00 3.00]
CPU.csp	[96.24 96.53 96.82]	[66.74 66.92 67.10]	[9.75 9.80 9.86]
Disk.csp	[147.48 148.41 149.34]	[100.49 101.08 101.67]	[21.00 21.17 21.34]
Loop.csp	[1578.87 1579.72 1580.58]	[930.45 930.95 931.45]	[277.30 277.45 277.60]
Oven.csp	[154.78 161.86 168.95]	[107.49 112.05 116.61]	[50.62 53.50 56.39]
ProdCons.csp	[1590.76 1591.41 1592.05]	[930.87 931.24 931.62]	[263.66 263.77 263.89]
ReadWrite.csp	[1489.52 1490.56 1491.60]	[1013.36 1014.03 1014.71]	[202.70 203.16 203.63]
Traffic.csp	[59.50 62.44 65.37]	[41.87 43.70 45.53]	[4.42 4.77 5.12]
Average	[559.59 568.46 577.37]	[352.21 358.25 364.27]	[89.72 91.54 92.77]

(b) Track size results

Benchmark	Memory Size (Bytes)	Total (ms)
ABP.csp	[11081.55 14972.37 18863.19]	[1311.01 1576.98 1842.95]
ATM.csp	[42118.79 47043.26 51967.73]	[627.51 687.70 747.88]
Buses.csp	[2487.00 2487.00 2487.00]	[126.87 127.57 128.28]
CPU.csp	[10138.70 10172.23 10205.76]	[200.40 201.36 202.31]
Disk.csp	[16517.61 16629.78 16741.95]	[221.42 222.36 223.32]
Loop.csp	[177843.37 177941.84 178040.32]	[2153.48 2154.34 2155.20]
Oven.csp	[19702.38 20642.42 21582.47]	[237.27 241.73 246.20]
ProdCons.csp	[167874.83 167944.98 168015.14]	[2162.36 2163.35 2164.35]
ReadWrite.csp	[161929.20 162170.57 162411.94]	[2169.92 2171.64 2173.36]
Traffic.csp	[6322.20 6640.38 6958.56]	[167.96 168.73 169.51]
Average	[61601.56 62664.48 63727.41]	[937.82 954.70 1005.34]

(c) Memory size and total time needed to produce a slice

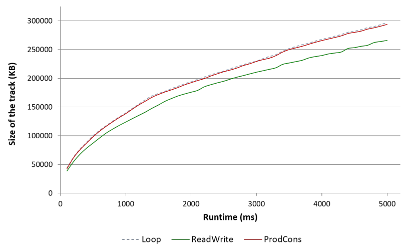
Table 1: Benchmark results showing CSP-Tracker performance

– `CSP2Erlang` is the time needed to compile the CSP program to an equivalent Erlang representation,

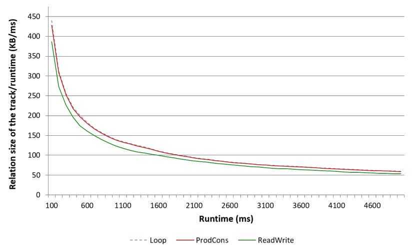
– `Runtime` is the time needed to execute the Erlang program (with a timeout of 2 seconds). The generation of the track is done by process `printer` (see Figure 7) in parallel (at the same time that the program is being executed). The time needed to generate the track is almost unappreciable (0.5% of the runtime),

– `Generate slice` is the time needed to slice the track and generate the final CSP slice, and

– `Total time` is the sum of the previous three measures. It represents the total amount of time needed by CSP-Tracker to execute a CSP benchmark, generate the track associated to the execution, and produce the final slice.



(a) Track size with respect to runtime



(b) Track growth rate with respect to runtime

Figure 8: Trend rate of the track growth.

- To measure memory performance we have:
  - `#Nodes` is the number of nodes that form the track,
  - `#Control Edges` is the number of control edges that form the track,
  - `#Sync. Edges` is the number of synchronization edges that form the track, and
  - `Memory Size`, the size in the hard disk of the track generated (stored as a .dot file).

In the tables we can see that computing slices is a very efficient task (less than 25 ms. in all cases). Of course, the total time depends on how long the computation is, but compilation from CSP to Erlang and slicing are of one order of magnitude less than execution. Regarding memory consumption, the biggest slice that can be computed is a slice that includes the whole track. Therefore, in column `Memory Size` we can see that it is as an average lower than 65 KB.

To study the trend rate of the track growth, we performed experiments with bigger timeouts and progressively measured the size of the track until we had enough data to approximate the cost function. This is shown in Figure 8(a). This figure shows a very similar growing rate for the three examples shown. As in the other experiments, in the three cases the result shown is the average of repeating the experiment 1000 times.

The function described in the figure is $y = x^{1/2}$, which corresponds to a polynomial (sublinear) cost. Therefore, if we measure the relation between the size of the graph and the runtime (which corresponds to the track generation speed), we get Figure 8(b). In this case, the function obtained is similar to $y = x^{-1/2}$. The interpretation of these figures is that the generation of the track slows down (less nodes are generated by unit of time) as the system becomes more complex (i.e., with more processes, more synchronizations, etc.).

The interested reader has available the web interface (`http://kaz.dsic.upv.es/csp_tracker`) with several already prepared examples to test the tool and its performance.

5.3. Dynamic slicing vs static slicing for CSP in practice

In this section we illustrate that our CSP dynamic slicer complements already existing CSP static slicers (see, e.g., [3, 20]). And, in particular, we show with an example how our dynamic slices are (in general) much more precise than the static slices produced by previous tools.

Consider the machine readable CSP specification in Figure 9 (ignore for the time being the underlining and the different colors). This specification defines a server that accepts different requests from clients: (i) request `getvalue` consults the server's state, (ii) request `stop` halts the server; and there are four types of requests that change the server's state: (iii) `inc` increases the state, (iv) `dec` decreases the state, (v) `reset` sets the state to zero, and (vi) `setvalue` sets the state to a given value. Four clients send in parallel their requests to the server. Clients `INC` and `DEC` are supposed to check the server's state before they change it. If the state is zero, they should stop the server. Probably, the reader has already noticed that `INC` behaves correctly, but `DEC` is buggy, because it does not take into account that other clients are concurrently changing the state of the server. Due to this bug, it is possible to produce a computation where the final state of the server is not zero when it is halted. We can easily see this situation with the track in Figure 10, where the difference between light and dark nodes can be ignored for the time being.

In this track, the last state of the server is `-1` (see node 60), which is a wrong state. We can select this node as the slicing criterion, and produce a dynamic slice to see what part of the code contributed to this wrong state. The dynamic slice in the track of Figure 10 is composed of the grey nodes. These nodes correspond to the black code in Figure 9. Note that only the code associated with clients `DEC` and `RESET` is part of the slice, because in this concrete execution the other clients do not participate.

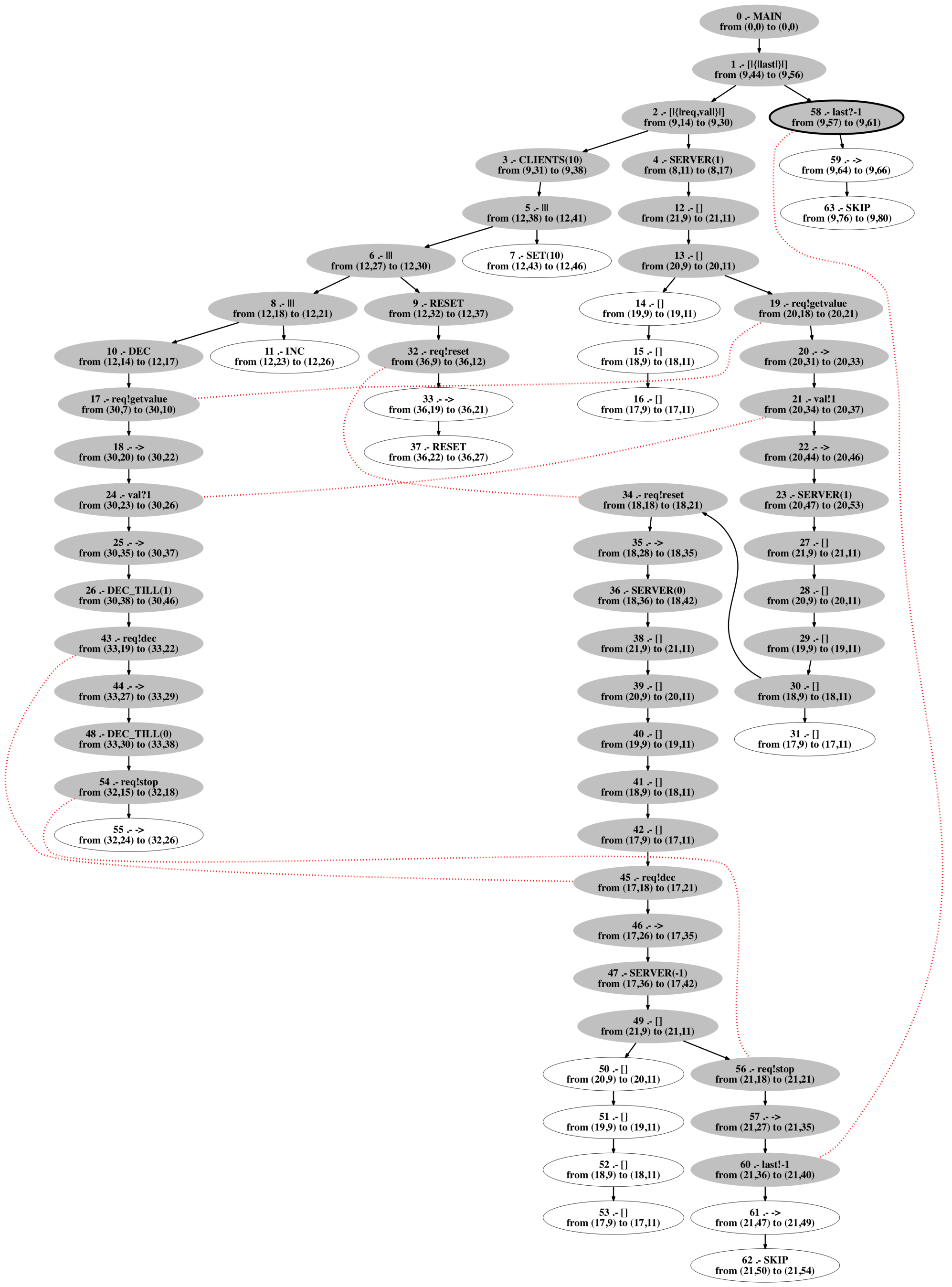
In contrast, if we use a static slicer with the same slicing criterion (i.e., event `last`), we get the slice shown in Figure 9 that is formed by the underlined code. This slice has been computed with the static slicer SOC [20]. Observe that a static slicer must consider all possible executions, and thus it includes code from the four clients, producing a much bigger slice (the dynamic slice is a subset of the static slice). Hence, if we want to analyze a particular execution, dynamic slices are more adequate and precise than their static counterparts.

6. Conclusions

This work defines the first adaptation of dynamic program slicing for CSP. The first main conclusion is that the traditional notion of slicing criterion is inappropriate for CSP, and thus we have proposed a new formulation of dynamic program slicing in the context of this language. The second main conclusion is that tracks are an ideal data structure to compute dynamic slices in CSP. Tracks represent a single execution and they contain all the information (including links to the source code) needed for slicing. Moreover, tracks allow us to compute slices in linear time.

```
-- Server's commands
channel inc, dec, reset, setvalue, getvalue, stop

channel req, val, last  -- Communication and output

-- Several clients send requests to a server in parallel
MAIN = (SERVER(1) [|{|req, val|}|] CLIENTS(10))

          [|{|last|}|]

          last?x -> SKIP

-- Four kinds of clients exist. They all are interleaved.
CLIENTS(x) = DEC ||| INC ||| RESET ||| SET(x)

-- The server has a state that can be changed by the clients' requests.
SERVER(state) = (req.inc -> SERVER(state+1))
                 [] (req.dec -> SERVER(state-1))
                 [] (req.reset -> SERVER(0))
                 [] (req.setvalue -> val?state -> SERVER(state))
                 [] (req.getvalue -> val!state -> SERVER(state))
                 [] (req.stop -> last!state -> SKIP)

-- Client INC increments the state of the server
INC = req!getvalue -> val?current ->
       if (current == 0)
       then req!stop -> SKIP
       else req!inc -> INC

-- Client DEC decrements the state of the server
DEC = req!getvalue -> val?current -> DEC_TILL(current)

DEC_TILL(0) = req!stop -> SKIP
DEC_TILL(other) = req!dec -> DEC_TILL(other-1)

-- Client RESET resets the state of the server to zero
RESET = req!reset -> RESET

-- Client SET(x) sets the state of the server to x
SET(x) = req!setvalue -> val!x -> SET(x)
```

Figure 9: CSP specification of a system with a server and four clients running in parallel



0 .- MAIN from (0,0) to (0,0)
1 .- [|{|last|}|] from (9,44) to (9,56)
2 .- [|{|req,val|}|] from (9,14) to (9,30)
58 .- last?-1 from (9,57) to (9,61)
3 .- CLIENTS(10) from (9,31) to (9,38)
4 .- SERVER(1) from (8,11) to (8,17)
59 .- -> from (9,64) to (9,66)
5 .- ||| from (12,38) to (12,41)
12 .- [] from (21,9) to (21,11)
63 .- SKIP from (9,76) to (9,80)
6 .- ||| from (12,27) to (12,30)
7 .- SET(10) from (12,43) to (12,46)
13 .- [] from (20,9) to (20,11)
8 .- ||| from (12,18) to (12,21)
9 .- RESET from (12,32) to (12,37)
14 .- [] from (19,9) to (19,11)
19 .- req!getvalue from (20,18) to (20,21)
10 .- DEC from (12,14) to (12,17)
11 .- INC from (12,23) to (12,26)
32 .- req!reset from (36,9) to (36,12)
15 .- [] from (18,9) to (18,11)
20 .- -> from (20,31) to (20,33)
17 .- req!getvalue from (30,7) to (30,10)
33 .- -> from (36,19) to (36,21)
16 .- [] from (17,9) to (17,11)
21 .- val!1 from (20,34) to (20,37)
18 .- -> from (30,20) to (30,22)
37 .- RESET from (36,22) to (36,27)
22 .- -> from (20,44) to (20,46)
24 .- val?1 from (30,23) to (30,26)
34 .- req!reset from (18,18) to (18,21)
23 .- SERVER(1) from (20,47) to (20,53)
25 .- -> from (30,35) to (30,37)
35 .- -> from (18,28) to (18,35)
27 .- [] from (21,9) to (21,11)
26 .- DEC_TILL(1) from (30,38) to (30,46)
36 .- SERVER(0) from (18,36) to (18,42)
28 .- [] from (20,9) to (20,11)
43 .- req!dec from (33,19) to (33,22)
38 .- [] from (21,9) to (21,11)
29 .- [] from (19,9) to (19,11)
44 .- -> from (33,27) to (33,29)
39 .- [] from (20,9) to (20,11)
30 .- [] from (18,9) to (18,11)
48 .- DEC_TILL(0) from (33,30) to (33,38)
40 .- [] from (19,9) to (19,11)
31 .- [] from (17,9) to (17,11)
54 .- req!stop from (32,15) to (32,18)
41 .- [] from (18,9) to (18,11)
55 .- -> from (32,24) to (32,26)
42 .- [] from (17,9) to (17,11)
45 .- req!dec from (17,18) to (17,21)
46 .- -> from (17,26) to (17,35)
47 .- SERVER(-1) from (17,36) to (17,42)
49 .- [] from (21,9) to (21,11)
50 .- [] from (20,9) to (20,11)
56 .- req!stop from (21,18) to (21,21)
51 .- [] from (19,9) to (19,11)
57 .- -> from (21,27) to (21,35)
52 .- [] from (18,9) to (18,11)
60 .- last!-1 from (21,36) to (21,40)
53 .- [] from (17,9) to (17,11)
61 .- -> from (21,47) to (21,49)
62 .- SKIP from (21,50) to (21,54)

Figure 10: Track associated with an execution of the code in Figure 9

In our theoretical formulation we redefined for CSP the notion of dynamic slice and dynamic slicing criterion. In particular, we propose the use of a particular occurrence of an event as the point of interest in the execution. This point of interest corresponds to exactly one node of the track associated with the computation. Events instead of variables are more appropriate for CSP given the event-based nature of this language. We also propose and discuss other variants of the slicing criterion that can be useful in different situations. These variants are novel and interesting because they imply considering various nodes in the track, or various terms in the source code.

Finally, we have implemented the first dynamic slicer for CSP based on tracks. It is publicly available and open-source.

References

[1] Agrawal, H., Horgan, J.R.: Dynamic Program Slicing. In: Fischer, B.N. (eds.) ACM SIGPLAN'90 Conf. on Programming Language Design and Implementation (PLDI), pp. 246–256. ACM, New York, NY, USA (1990)

[2] Binkley, D., Gallagher, K.B.: Program Slicing. Advances in Computers 43, 1–50 (1996)

[3] Brückner, I.: Slicing Concurrent Real-Time System Specifications for Verification. In: Davies, J. and Gibbons, J. (eds.) IFM 2007. LNCS, vol. 4591, pp. 54–74. Springer, Heidelberg (2007)

[4] Brückner, I., Wehrheim, H.: Slicing an Integrated Formal Method for Verification. In: Lau, K.K., Banach, R. (eds.) ICFEM 2005. LNCS, vol. 3785, pp. 360–374. Springer, Heidelberg (2005)

[5] Chitil, O.: A Semantics for Tracing. In: Arts, T., Mohnen, M. (eds.) 13th Int'l Workshop on Implementation of Functional Languages (IFL'01), pp. 249–254. Ericsson Computer Science Laboratory, Sweden (2001)

[6] Danicic, S., Harman, M.: Program Slicing Using Functional Networks (invited paper). In: Ulidowski, I. (ed.) 4th. RIMS Workshop on Concurrency Theory and Applications, pp. 54–65. Kyoto University, Japan (1996)

[7] Díaz, J., Muñoz-Caro, C., Niño, A.: A Survey of Parallel Programming Models and Tools in the Multi and Many-Core Era. IEEE Transactions on Parallel and Distributed Systems 23(8), 1369–1386 (2012)

[8] Ferrante, J., Ottenstein, K.J., Warren, J.D.: The Program Dependence Graph and Its Use in Optimization. ACM Transactions on Programming Languages and Systems 9(3), 319–349 (1987)

[9] Georges, A., Buytaert, D., Eeckhout, L.: Statistically Rigorous Java Performance Evaluation. In: Gabriel, R.P., Bacon, D.F., Lopes, C.V., Jr., G.L.S.

(eds.) 22nd Annual ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'07), pp. 57–76. ACM, New York, USA (2009)

[10] Gibson-Robinson, T., Armstrong, P. J., Boulgakov, A., Roscoe,A. W.: FDR3 - A Modern Refinement Checker for CSP. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 187–201. Springer, Heildeberg (2014)

[11] Heimdahl, M., Whalen, M.: Reduction and Slicing of Hierarchical State Machines. In: Jazayeri, M., Schauer, H. (eds.) ESEC/FSE 1997. LNCS, vol. 1301, pp. 450–467. Springer, Heildeberg (1997)

[12] Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1985)

[13] Kavi, K.M., Sheldon, F.T., Shirazi, B., Hurson, A.R.: Reliability Analysis of CSP Specifications using Petri Nets and Markov Processes. In: 28th Annual Hawaii Int'l Conf. on System Sciences (HICSS-28), vol. 2 (Software Technology), pp. 516–524. IEEE Computer Society, Washington, DC, USA (1995)

[14] Khan, Y.I., Guelfi, N.: Survey of Petri nets Slicing. Tech. rep., Computer Science and Communications Research Unit, Faculty of Science, Technology and Communication, University of Luxembourg, Luxembourg (2013)

[15] Korel, B., Laski, J.W.: Dynamic Program Slicing. Information Processing Letters 29(3), 155–163 (1988)

[16] Krinke, J.: Context-Sensitive Slicing of Concurrent Programs. In: 11th ACM SIGSOFT Symposium on Foundations of Software Engineering (ESEC/FSE'03), pp. 178–187. ACM, New York, NY, USA (2003)

[17] Ladkin, P., Simons, B.: Static Deadlock Analysis for CSP-Type Communications. In: Fussell, D., Malek, M. (eds.) Responsive Computer Systems: Steps Toward Fault-Tolerant Real-Time Systems, Chapter 5, Kluwer Academic Publishers (1995)

[18] Leuschel, M., Butler, M.J.: ProB: an Automated Analysis Toolset for the B Method. Journal of Software Tools for Technology Transfer 10(2), 185–203 (2008)

[19] Leuschel, M., Falampin, J., Fritz, F., Plagge, D.: Automated Property Verification for Large Scale B Models with ProB. Formal Aspects of Computing 23(6), 683–709 (2011)

[20] Leuschel, M., Llorens, M., Oliver, J., Silva, J., Tamarit, S.: SOC: a Slicer for CSP Specifications. In: Puebla, G., Vidal, G. (eds.) 2009 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM'09), pp. 165–168. ACM, New York, NY, USA (2009)

[21] Leuschel, M., Llorens, M., Oliver, J., Silva, J., Tamarit, S.: Static Slicing of Explicitly Synchronized Languages. Information and Computation 214, 10–46 (2012)

[22] Llorens, M., Oliver, J., Silva, J., Tamarit, S.: A Tracking Semantics for CSP. In: Bolduc, C., Desharnais, J., Ktari, B. (eds.) MPC 2010. LNCS, vol. 6120, pp. 248–270. Springer, Heildeberg (2010)

[23] Lowe, G.: Breaking and Fixing the Needham-Schroeder Public-Key Protocol Using FDR. In: Margaria, T., Steffen, B. (eds.) TACAS 1996. LNCS, vol. 1055, pp. 147–166. Springer, Heidelberg (1996)

[24] Ochoa, C., Silva, J., Vidal, G.: Dynamic slicing based on redex trails. Proceedings of the 2004 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation (PEPM'04). pp. 123–134. ACM, Verona, Italy (2004)

[25] Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (2005)

[26] Silva, J.: A Vocabulary of Program Slicing-Based Techniques. ACM Computing Surveys 44(3), 12 (2012)

[27] Sloane, A., Holdsworth, J.: Beyond Traditional Program Slicing. In: Int'l Symp. on Software Testing and Analysis, pp. 180–186. ACM Press, San Diego, CA (1996)

[28] Souilah, I., Francalanza, A., Sassone, V.: A Formal Model of Provenance in Distributed Systems. In: Cheney, J. (ed.) First Workshop on the Theory and Practice of Provenance (TaPP'09), pp. 1–11. USENIX Association (2009)

[29] Tallam, S., Tian, C., Gupta, R.: Dynamic Slicing of Multithreaded Programs for Race Detection. In: 24th IEEE International Conference on Software Maintenance (ICSM'08), pp. 97–106. IEEE Computer Society (2008)

[30] Tip, F.: A Survey of Program Slicing Techniques. Journal of Programming Languages 3(3), 121–189 (1995)

[31] Weeratunge, D., Zhang, X., Sumner, W.N., Jagannathan, S.: Analyzing Concurrency Bugs Using Dual Slicing. In: Tonella, P., Orso, A. (eds.) 19th Int'l Symposium on Software Testing and Analysis (ISSTA'10), pp. 253–264. ACM, New York, USA (2010)

[32] Weiser, M.: Program Slicing. IEEE Transactions on Software Engineering 10(4), 352–357 (1984)

[33] Zhao, X., Jamali, N.: Supporting Deadline Constrained Distributed Computations on Grids. In: Jha, S., gentschen Felde, N., Buyya, R., Fedak, G. (eds.) 12th IEEE/ACM Int'l Conf. on Grid Computing (GRID'11), pp. 165–172. IEEE Computer Society (2011)

Appendix A. The syntax and semantics of CSP

We recall in this appendix the syntax of CSP. Figure A.11 summarizes the syntax constructs used in CSP specifications. A *specification* is viewed as a finite set of process definitions. The left-hand side of each definition is the name of a process, which is defined in the right-hand side (abbrev. *rhs*) by means of an expression that can be a call to another process or a combination of the following operators:

(Prefixing) It specifies that the compound object CO must happen before process P. Compound objects represent events and communications.

(Internal choice) The system chooses non-deterministically to execute one of the two processes P or Q.

(External choice) It is identical to internal choice but the choice comes from the external environment (e.g., the user).

(Conditional choice) It is a choice that depends on a condition, i.e., it is equivalent to `if` $Bool$ `then` P `else` Q.

(Sequential composition) It specifies a sequence of two processes. If the first one (successfully) finishes, the second can start.

(Synchronized parallelism) Both processes are executed in parallel with a set $\{\overline{EV_n}\}$ of synchronized events. In absence of synchronizations both processes can execute in any order. Whenever a synchronized event $a \in \{\overline{EV_n}\}$ happens in one of the processes it must also happen in the other at the same time. Whenever the set of synchronized events is not specified, it is assumed that processes are synchronized in all common events. A particular case of parallel execution is *interleaving* where no synchronizations exist (i.e., $\{\overline{EV_n}\} = \emptyset$). It is often denoted with the operator $|||$.

(Hiding) Process P is executed with a set of hidden events $\{\overline{EV_n}\}$. Hidden events are not observable from outside the process, and thus, they cannot synchronize with other processes.

(Renaming) Process P is executed with a set of renamed events specified with the total mapping $\Re$. An event a renamed as b behaves internally as a but it is observable as b from outside the process.

(Skip) It is a process that successfully finishes. It allows us to continue the next sequential process if any.

(Stop) Synonymous with deadlock. It is a process that finishes and it does not allow the next sequential process to continue if any.

The domain Σ of events contains basic symbols such as a that can be compounded to produce communications:

(Input) It is used to receive a message from another process. For instance, if $A \subseteq \Sigma$ is any set of events and, for each $x \in A$, we have defined a process $P(x)$, then $c?x : A \rightarrow P(x)$ defines the process which accepts any element a of A and then behaves like the appropriate $P(a)$.

(Output) It is complementary to the input. In this case, $c!x$ is used to send message x.

We allow events that have been constructed out of any finite number of parts using the infix dot '.' (which is assumed to be associative), e.g., $c.a$.

Domains

$M, N \ldots \in \mathcal{N}$	(Process names)
$P, Q \ldots \in \mathcal{P}$	(Processes)
$a, b \ldots \in \Sigma$	(Events)
$x, y \ldots \in \Sigma_{\mathcal{V}}$	(Events with variables)

S	$::=$	$\{D_1, \ldots, D_n\}$	(Entire specification)	
D	$::=$	$N = P$	(Process definition)	
	$\mid$	$N(\overline{EV_n}) = P$	(Parameterized process)	$\overline{EV_n} = EV_1, \ldots, EV_n$
P	$::=$	M	(Process call)	
	$\mid$	$M(\overline{EV_n})$	(Parameterized process call)	
	$\mid$	$CO \rightarrow P$	(Prefixing)	
	$\mid$	$P \sqcap Q$	(Internal choice)	
	$\mid$	$P \Box Q$	(External choice)	
	$\mid$	$P \nless Bool \ngtr Q$	(Conditional choice)	
	$\mid$	$P \,;\, Q$	(Sequential composition)	
	$\mid$	$P \underset{\{\overline{EV_n}\}}{\Vert} Q$	(Synchronized parallelism)	
	$\mid$	$P \backslash \{\overline{EV_n}\}$	(Hiding)	
	$\mid$	$P[\![\Re]\!]$	(Renaming)	$\Re \subseteq \Sigma_{\mathcal{V}} \times \Sigma_{\mathcal{V}}$
	$\mid$	$SKIP$	(Skip)	
	$\mid$	$STOP$	(Stop)	
CO	$::=$	$EV \mid CO?EVI \mid CO!EV$	(Compound Object)	
EVI	$::=$	$EV \mid v : T$	(Input event with Variables)	$T \subseteq \Sigma$, v is a variable
EV	$::=$	$a \mid v \mid EV.EV$	(Event with Variables)	v is a variable
$Bool$	$::=$	$true \mid false \mid Bool \vee Bool$	(Boolean expression)	
	$\mid$	$Bool \wedge Bool \mid \neg Bool$		
	$\mid$	$EV = EV \mid EV \neq EV$		

Figure A.11: Syntax of CSP specifications

We now recall the standard operational semantics of CSP as defined by A.W. Roscoe [25]. It is presented in Figure A.12 as a logical inference system. A *state* of the semantics is a process to be evaluated called the *control*. The inference system starts with an initial state, and the rules of the semantics are used to infer how this state evolves. When no rules can be applied to the current state, the computation finishes. The rules of the semantics change the states of the computation due to the occurrence of events. The set of possible events is $\Sigma \cup \{\tau, \checkmark\}$. Events in $\Sigma = \{a, b, \ldots\}$ are visible from the external environment, and can only happen with its co-operation (e.g., actions of the user). The special event τ cannot be observed from outside the system and it is an internal event that happens automatically as defined by the semantics. $\checkmark$ is a special event representing the successful termination of a process. We use the special symbol Ω to denote any process that successfully terminated.

In order to perform computations, we construct an initial state (e.g., `MAIN`) and (non-deterministically) apply the rules of Figure A.12. The intuitive meaning of each rule is the following:

((Parameterized) Process Call) In a process call, the call is unfolded and the right-hand side of process N becomes the new control. In a parameterized process call, the behavior is the same, but in this case we use function *subs* to substitute in $rhs(N)$ all variables $\overline{x_n}$ by the actual values of the parameters $\overline{a_n}$.

(Prefixing) When event co occurs, process P becomes the new control. The only way communications are introduced into the operational semantics is via the prefixing operation $co \rightarrow P$. In general, co may be a compound object, perhaps involving much computation to work out what it represents. The prefix co may represent a range of possible communications and bind one or more identifiers in P. $comms(co)$ is the set of communications described by co. We deal only with closed terms: processes with no free identifiers. Using this, it is possible to handle most of the situations that can arise, making sure that each identifier has been substituted by a concrete value by the time we need to know it. For $a \in comms(co)$, $subs(a, co, P)$ is the result of substituting the appropriate part of a for each identifier in P bound by co. This equals P if there are no identifiers bound.

(SKIP) After `SKIP`, the only possible event is $\checkmark$, which denotes the successful termination of the (sub)computation with the special symbol Ω. There is no rule for Ω (neither for `STOP`), hence, this (sub)computation has finished.

(Internal Choice 1 and 2) The system, with the occurrence of the internal event τ, (non-deterministically) selects one of the two processes P or Q which is the new control.

(External Choice 1, 2, 3 and 4) The occurrence of τ develops one of the branches. The occurrence of an event $e \neq \tau$ is used to select one of the two processes P or Q and the control changes according to the event.

(Process Call)

$$\frac{}{N \xrightarrow{\tau} rhs(N)}$$

(Parameterized Process Call)

$$\frac{}{N(\overline{a_n}) \xrightarrow{\tau} subs(\overline{a_n}, \overline{x_n}, rhs(N))}$$

where $N(\overline{x_n}) = rhs(N) \in \mathcal{S}$
with $\overline{x_n} \in \Sigma_{\mathcal{V}} \wedge \overline{a_n} \in \Sigma$

(Prefixing)

$$\frac{}{(co \to P) \xrightarrow{a} subs(a, co, P)} \quad a \in comms(co)$$

(SKIP)

$$\frac{}{\texttt{SKIP} \xrightarrow{\checkmark} \Omega}$$

(Internal Choice 1)

$$\frac{}{(P \sqcap Q) \xrightarrow{\tau} P}$$

(Internal Choice 2)

$$\frac{}{(P \sqcap Q) \xrightarrow{\tau} Q}$$

(External Choice 1)

$$\frac{P \xrightarrow{\tau} P'}{(P \;\Box\; Q) \xrightarrow{\tau} (P' \;\Box\; Q)}$$

(External Choice 2)

$$\frac{Q \xrightarrow{\tau} Q'}{(P \;\Box\; Q) \xrightarrow{\tau} (P \;\Box\; Q')}$$

(External Choice 3)

$$\frac{P \xrightarrow{e} P'}{(P \;\Box\; Q) \xrightarrow{e} P'} \quad e \in \Sigma \cup \{\checkmark\}$$

(External Choice 4)

$$\frac{Q \xrightarrow{e} Q'}{(P \;\Box\; Q) \xrightarrow{e} Q'} \quad e \in \Sigma \cup \{\checkmark\}$$

(Conditional Choice 1)

$$\frac{}{(P \not< Bool \not> Q) \xrightarrow{\tau} P} \quad \text{if } Bool = true$$

(Conditional Choice 2)

$$\frac{}{(P \not< Bool \not> Q) \xrightarrow{\tau} Q} \quad \text{if } Bool = false$$

(Sequential Composition 1)

$$\frac{P \xrightarrow{e} P'}{(P; Q) \xrightarrow{e} (P'; Q)} \quad e \in \Sigma \cup \{\tau\}$$

(Sequential Composition 2)

$$\frac{P \xrightarrow{\checkmark} \Omega}{(P; Q) \xrightarrow{\tau} Q}$$

(Synchronized Parallelism 1)

$$\frac{P \xrightarrow{e'} P'}{(P \underset{X}{||} Q) \xrightarrow{e} (P' \underset{X}{||} Q)} \quad \begin{array}{l}(e = e' \in \Sigma \backslash X) \vee \\ (e = \tau \;\wedge\; e' \in \{\tau, \checkmark\})\end{array}$$

(Synchronized Parallelism 2)

$$\frac{Q \xrightarrow{e'} Q'}{(P \underset{X}{||} Q) \xrightarrow{e} (P \underset{X}{||} Q')} \quad \begin{array}{l}(e = e' \in \Sigma \backslash X) \vee \\ (e = \tau \;\wedge\; e' \in \{\tau, \checkmark\})\end{array}$$

(Synchronized Parallelism 3)

$$\frac{P \xrightarrow{a} P' \qquad Q \xrightarrow{a} Q'}{(P \underset{X}{||} Q) \xrightarrow{a} (P' \underset{X}{||} Q')} \quad a \in X$$

(Synchronized Parallelism 4)

$$\frac{}{(\Omega \underset{X}{||} \Omega) \xrightarrow{\checkmark} \Omega}$$

(Hiding 1)

$$\frac{P \xrightarrow{a} P'}{(P \backslash B) \xrightarrow{\tau} (P' \backslash B)} \quad a \in B$$

(Hiding 2)

$$\frac{P \xrightarrow{e} P'}{(P \backslash B) \xrightarrow{e} (P' \backslash B)} \quad (e \in \Sigma \wedge e \notin B) \vee (e = \tau)$$

(Hiding 3)

$$\frac{P \xrightarrow{\checkmark} \Omega}{(P \backslash B) \xrightarrow{\checkmark} \Omega}$$

(Renaming 1)

$$\frac{P \xrightarrow{e'} P'}{(P[\![\Re]\!]) \xrightarrow{e} (P'[\![\Re]\!])} \quad \begin{array}{l}(e, e' \in \Sigma \wedge e' \;\Re\; e) \;\vee \\ (e = e' = \tau)\end{array}$$

(Renaming 2)

$$\frac{P \xrightarrow{\checkmark} \Omega}{(P[\![\Re]\!]) \xrightarrow{\checkmark} \Omega}$$

Figure A.12: CSP's operational semantics

(Conditional Choice 1 and 2) The condition $Bool$ is evaluated. If it is $true$, process P is put in the control, if it is $false$, process Q is.

(Sequential Composition 1) In $P; Q$, P can evolve to P' with any event except $\checkmark$. Hence, the control becomes $P'; Q$.

(Sequential Composition 2) When P successfully finishes (with event $\checkmark$), Q can start. Note that $\checkmark$ is hidden from outside the whole process becoming τ.

(Synchronized Parallelism 1 and 2) When an event $e \notin X$ or events τ or $\checkmark$ occur in a branch, the corresponding process (either P or Q) evolves accordingly. Note that $\checkmark$ is hidden from outside the whole process becoming τ.

(Synchronized Parallelism 3) When a visible event $a \in X$ happens, it is required that both processes synchronize, P and Q are executed at the same time and the control becomes $P' \underset{X}{||} Q'$.

(Synchronized Parallelism 4) When both processes have successfully terminated the control becomes Ω and the event $\checkmark$ is visible from outside.

(Hiding 1 and Hiding 2) When event $a \in B$ ($B \subseteq \Sigma$) occurs in P, it is hidden, and thus changed to τ so that it is not observable from outside P. Contrarily, when event $a \notin B$ occurs in P, it behaves normally.

(Hiding 3) When P finishes ($\checkmark$ happens), the control becomes Ω.

(Renaming 1) Whenever an event a happens in P, it is renamed to b ($a \Re b$) so that, externally, only b is visible. Renaming has no effect on events renamed to themselves ($a \Re a$), τ and $\checkmark$.

(Renaming 2) When P finishes ($\checkmark$ happens), the control becomes Ω.