Tracking CSP computations[☆]

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

Departamento de Sistemas Informáticos y Computación
Universitat Politècnica de València
Valencia, Spain

Abstract

Tracing is one of the most important techniques for program understanding and debugging. A trace gives the user access to otherwise hidden information about a computation. In the context of concurrent languages, computations are particularly complex due to the non-deterministic execution order of processes and to the restrictions imposed on this order by synchronizations; hence, a tracer is a powerful tool to explore, understand and debug concurrent computations. In CSP, traces are sequences of events that define a particular execution. This notion of trace is completely different to the one used in other paradigms where traces are formed by those source code expressions evaluated during a particular execution. We refer to this second notion of traces as tracks. In this work, we introduce the theoretical basis for tracking concurrent and explicitly synchronized computations in process algebras such as CSP. Tracking computations in this kind of systems is a difficult task due to the subtleties of the underlying operational semantics which combines concurrency, non-determinism and non-termination. We define an instrumented operational semantics that generates as a side-effect an appropriate data structure (a track) which can be used to track computations. The formal definition of a tracking semantics improves the understanding of the tracking process, but also, it allows us to formally prove the correctness of the computed tracks.

Keywords: Concurrency theory/modelling; CSP; Semantics; Tracking.

1. INTRODUCTION

One of the most widespread concurrent specification languages is the *Communicating Sequential Processes* (CSP) [22, 44] whose operational semantics allows the combination of parallel, non-deterministic and non-terminating processes. CSP allows for a precise description and analysis of event-based concurrency. An important advantage of this language is that it offers a well-developed syntax, algebraic and operational semantics, a hierarchy of congruent denotational semantic models, as well as a formal theory of refinement and compositional verification [43].

The study and transformation of CSP specifications often uses different analyses such as deadlock analysis [28], reliability analysis [25], security analysis [37] and program slicing [56], which are based on a data structure able to represent computations through the use of traces [11]. Two of the most important analysis tools for CSP are ProB [30, 31] and FDR [16]. Both are refinement checking software tools, designed to check formal models. Specifically, they implement an animator, constraint solver and model checker for CSP. ProB is now being used within Siemens, Alstom, and several other companies for data

[☆]This work has been partially supported by the EU (FEDER) and the *Spanish Ministerio de Ciencia, Innovación y Universidades*/AEI under grant TIN2016-76843-C4-1-R and by the *Generalitat Valenciana* under grant PROMETEO-II/2015/013 (SmartLogic). The authors acknowledge a partial support of European COST Action IC1405 on Reversible Computation - extending horizons of computing. Salvador Tamarit was partially supported by the *Conselleria de Educación, Investigación, Cultura y Deporte de la Generalitat Valenciana* under the grant APOSTD/2016/036.

[*]Corresponding author

Email addresses: `mllorens@dsic.upv.es` (M. Llorens), `fjoliver@dsic.upv.es` (J. Oliver), `jsilva@dsic.upv.es` (J. Silva), `stamarit@dsic.upv.es` (S. Tamarit) validation.[1] FDR, originally developed by Formal Systems (Europe) Ltd, is, since 2008, developed in Oxford University under support from EPSRC, ONR, and industry. The last release of FDR is FDR4, whose first stable version appeared in late 2016.

CSP has been used in research, teaching, and industry [1, 4, 29, 42], and is nowadays applied to many different industrial problems such as medical simulations [35, 20], modeling web services choreographies [34], hardware simulation and verification [38, 19], cloud computing models [23], widespread concurrent languages (e.g. MPI [9], Go [53] and Verilog [48]), testing [10], and social networks modeling [24, 41].

Even though traces are always used to represent computations, the word *trace* is polysemic, and its exact meaning changes depending on the programming language we are referring to. For instance, in languages such as Haskell, traces are graph-like data structures formed by the redexes that are evaluated in a computation. Traces such as the *Augmented Redex Trail* are the basis of many analysis methods and tools (see, e.g., [11, 12, 13, 3]). In other languages the notion of trace is essentially different. Concretely, in CSP a trace is not a graph, but a sequence of events (see Chapter 8 of [44] for a detailed study of this kind of traces). To distinguish both notions of trace, we call the first kind of traces—not existing in CSP—*tracks*, and we use the word *trace* for its standard meaning in CSP (i.e., sequence of events).

In this work we introduce tracks in CSP. In our setting, a track is a data structure that represents the sequence of expressions that have been evaluated during the computation, and moreover, this data structure is labeled with the location of these expressions in the specification. Therefore, a CSP track is much more informative than a CSP trace since the former not only contains a lot of information about original program structures but also explicitly relates the sequence of events with the parts of the specification that caused these events.

Example 1. *Consider the finite state machine (*FSM*) in Figure 1 used to recognize strings of a language. The following CSP processes*[2] *describe the previous finite state machine (for the time being ignore the different colours and the box). Event* `valid` *(respectively* `notvalid`*) is used to denote that a string has (respectively not) been recognized. The specification is composed of three main processes that run in parallel and synchronize when needed:* `FSM`, `INPUT`*, and* `CHECK`*. Process* `FSM` *models the finite state machine. It uses a parameter to indicate the current state, for instance,* `FSM(1)` *indicates that the finite state machine is currently in state* `s1`*. Process* `INPUT` *represents the string to evaluate, and process* `CHECK` *determines whether this string is valid or not.*

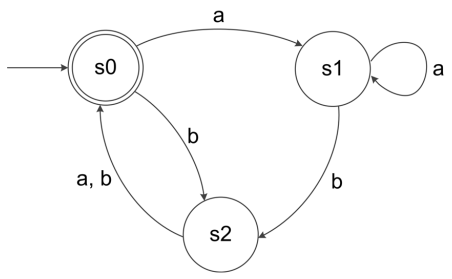


Figure 1: FSM used to recognize tokens

$$\mathtt{MAIN} = ((\mathtt{FSM(0)} \underset{\{a,b\}}{\|} \mathtt{INPUT}) \underset{\{a,b\}}{\|} \mathtt{CHECK(s0)}) \backslash \{\mathtt{end}\}$$

$$\mathtt{FSM(0)} = \mathtt{a!s1} \rightarrow \mathtt{FSM(1)} \,\square\, \mathtt{b!s2} \rightarrow \mathtt{FSM(2)}$$

$$\mathtt{FSM(1)} = \mathtt{a!s1} \rightarrow \mathtt{FSM(1)} \,\square\, \mathtt{b!s2} \rightarrow \mathtt{FSM(2)}$$

$$\mathtt{FSM(2)} = \mathtt{a!s0} \rightarrow \mathtt{FSM(0)} \,\square\, \mathtt{b!s0} \rightarrow \mathtt{FSM(0)}$$

$$\mathtt{INPUT} = \mathtt{a?state1} \rightarrow \boxed{\mathtt{b?state2}} \rightarrow \mathtt{b?state3} \rightarrow \mathtt{a?state4} \rightarrow \mathtt{b?state5} \rightarrow \mathtt{end!state5} \rightarrow \mathtt{STOP}$$

$$\mathtt{CHECK(fin)} = \mathtt{end?st} \rightarrow ((\mathtt{valid} \rightarrow \mathtt{SKIP}) \not< \mathtt{st} = \mathtt{fin} \not> (\mathtt{notvalid} \rightarrow \mathtt{SKIP}))$$

[1]Specific information about the industrial use of ProB can be found at `https://www3.hhu.de/stups/prob/index.php/`.
[2]We refer those readers non familiar with CSP syntax to Section 3 where we provide a brief introduction to CSP.

The whole specification models a FSM not accepting the string `abbab`*. The trace produced by this specification is* $\langle$`a.s1`, `b.s2`, `b.s0`, `a.s1`, `b.s2`, `notvalid`$\rangle$*. If we observe the track in Figure 2 (for the time being ignore the different shapes and colours of nodes) it is possible to identify the processes and their synchronizations, which are represented with dashed arcs. Each node represents a term in the source code in a specific instant of the computation, i.e., each time a term is evaluated it is represented in the track with a new node. Arcs are of two types: control-flow arcs and synchronizations. Control-flow arcs somehow represent a timeline. They represent the transition from one term to another term during the evaluation of the specification. Synchronizations always connect nodes whose associated term is a prefix. All this provides some interesting properties: (i) one can follow control-flow arcs to know the order in which source code terms where evaluated. (ii) One node represents one specific term in one specific evaluation instant. (iii) 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 prefixes in a path of synchronization arcs must occur at the same time.*

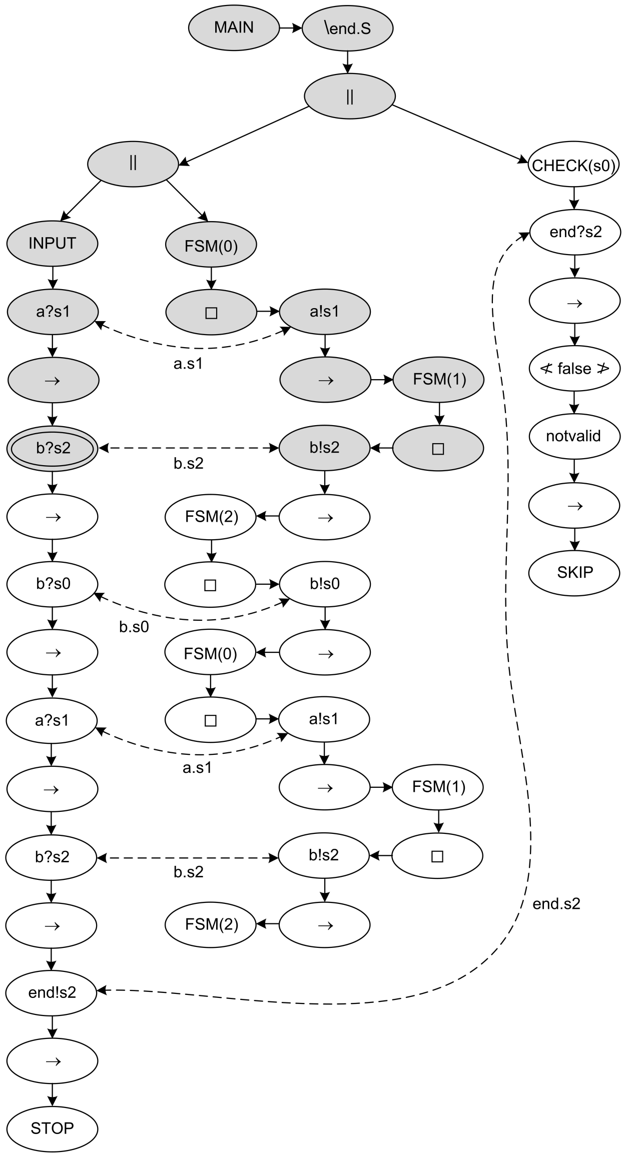


Figure 2: Track of the program in Example 1

1.1. Motivation

Tracks can be useful for several applications, including debugging, program comprehension, and component extraction. In this section we illustrate some of these applications with examples.

1.1.1. Debugging

In Figure 2, we can observe that the track contains explicit information about the specification expressions that were involved in the computation.[3] Therefore, it can be used for dynamic program slicing (see [54, 49] for an explanation of the technique and [33] for an adaptation of static program slicing to CSP). In particular, in Example 1, we can use the track to extract the part of the program that was involved in a particular sub-computation. For instance, if we select in the track the first occurrence of event `b` in process `INPUT` (this node has a double line), we can identify the part of the track that was necessarily executed before the occurrence of this event. This part has been coloured in grey. Of course, we can automatically map the coloured parts of the track to the specification and discover the part of the specification that is needed to execute event `b` (it is marked with a box in the specification) for the first time. This part has been highlighted in black in the example. With a quick look, one can see that the underlined parts skip process `CHECK`. This is not only useful for program comprehension, it is also useful for other program analyses and can help to speed up the processing or analysis of a specification by only concentrating on the part that affects the event or expression of interest (see, e.g., [7]).

We now show a more detailed example that illustrates the usefulness of tracks for debugging.

We consider a CSP specification used to simulate the process scheduler of a CPU. The CPU contains three main components (see Figure 3): an Arithmetic Logic Unit (ALU), a Control Unit (CU), and a process 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.

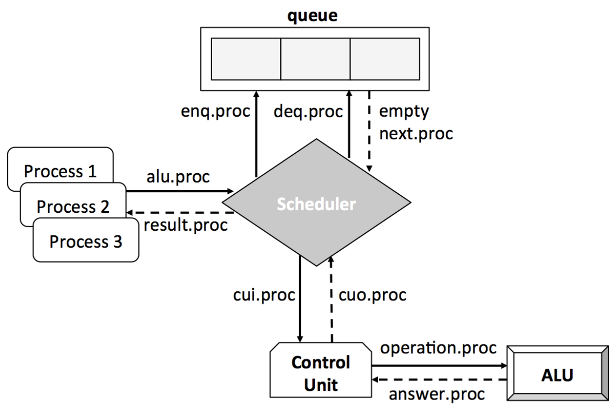
This system can be modeled with the CSP specification at the bottom of the figure. The specification is buggy. It is syntactically correct, but the traces produced are not the expected ones. In particular, one can generate the trace $\langle \texttt{alu.3}, \texttt{working.3}, \texttt{cui.3}, \texttt{result.3}, \texttt{operation.3} \rangle$, which 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 ($\texttt{result.3}, \texttt{operation.3}$) 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 event `result.X` of process `Process(X)` as the slicing criterion. CSP-Tracker[4] includes the first dynamic slicer for CSP. It can automatically extract the dynamic slice produced for that slicing criterion using the internal track generated. The slice only contains the black code, which is enough to produce the error. Thus, it must contain a bug.

For the computation of slices we use tracks. Note that they are an internal data structure, and thus the user does not need to inspect the track, just the sliced source code. For instance, a part of the track associated with the computation that produced the bug is depicted in Figure 4. The code that must be inspected in the slice is very small compared to the original specification. But, the track can still help us to inspect the slice to discover the bug because it allows us to graphically inspect the code at the same time that we follow the trace.

[3]For the sake of clarity, we have omitted the identifiers that point to unique expressions in the source code. Instead, we have included the referenced expression.

[4]CSP-Tracker is the tracker tool that the authors developed following the ideas presented in this paper. Section 7 provides implementation details and describes its functionality.



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

MAIN = SYSTEM

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

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

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

Sched = Sched_idle

Sched_idle = alu?proc $\rightarrow$ cui!proc $\rightarrow$ Sched_busy

Sched_busy = cuo?proc $\rightarrow$ (result.proc $\rightarrow$ Sched_check) $\square$ alu?proc $\rightarrow$ (enq!proc $\rightarrow$ Sched_busy)

Sched_check = deq $\rightarrow$ (empty $\rightarrow$ Sched_idle $\square$ next?proc $\rightarrow$ cui!proc $\rightarrow$ Sched_busy)

CU = cui?proc $\rightarrow$ operation!proc $\rightarrow$ answer?proc $\rightarrow$ cuo!proc $\rightarrow$ CU

ALU = operation?proc $\rightarrow$ answer!proc $\rightarrow$ ALU

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

DQ(2) = deq $\rightarrow$ shift $\rightarrow$ X(2)

DQ(i) = enq?x $\rightarrow$ (left!x $\rightarrow$ shift $\rightarrow$ DQ(i+1)) $\square$ deq $\rightarrow$ (empty $\rightarrow$ DQ(0) $\not\prec$ i==0 $\not\succ$ X(i))

X(i) = right?y $\rightarrow$ (next!y $\rightarrow$ DQ(i-1)) $\square$ shift $\rightarrow$ X(i)

BUFF = (CELL $[\![$right $\Re$ comm$]\!]$ $\underset{\{\text{comm}\}}{\|}$ CELL $[\![$left $\Re$ comm$]\!]$)\{comm}

CELL = left?x $\rightarrow$ shift $\rightarrow$ right!x $\rightarrow$ CELL

Figure 3: A (buggy) specification in CSP of a CPU diagram

In Figure 4, one can follow the trace and very quickly see that event `result.3` (node 32)[5] is not synchronized (there is not a dashed arc connecting this node). This means that this event occurred internally, without any communication. However, looking at Figure 3, we can see that the processes and the scheduler communicate via two messages: `alu.proc` and `result.proc`. Therefore, it is clear that

[5]The notation "`from (9,36) to (9,42)`" specifies the starting and ending positions of the term in the source code, i.e., from line 9, column 36 to line 9, column 42.

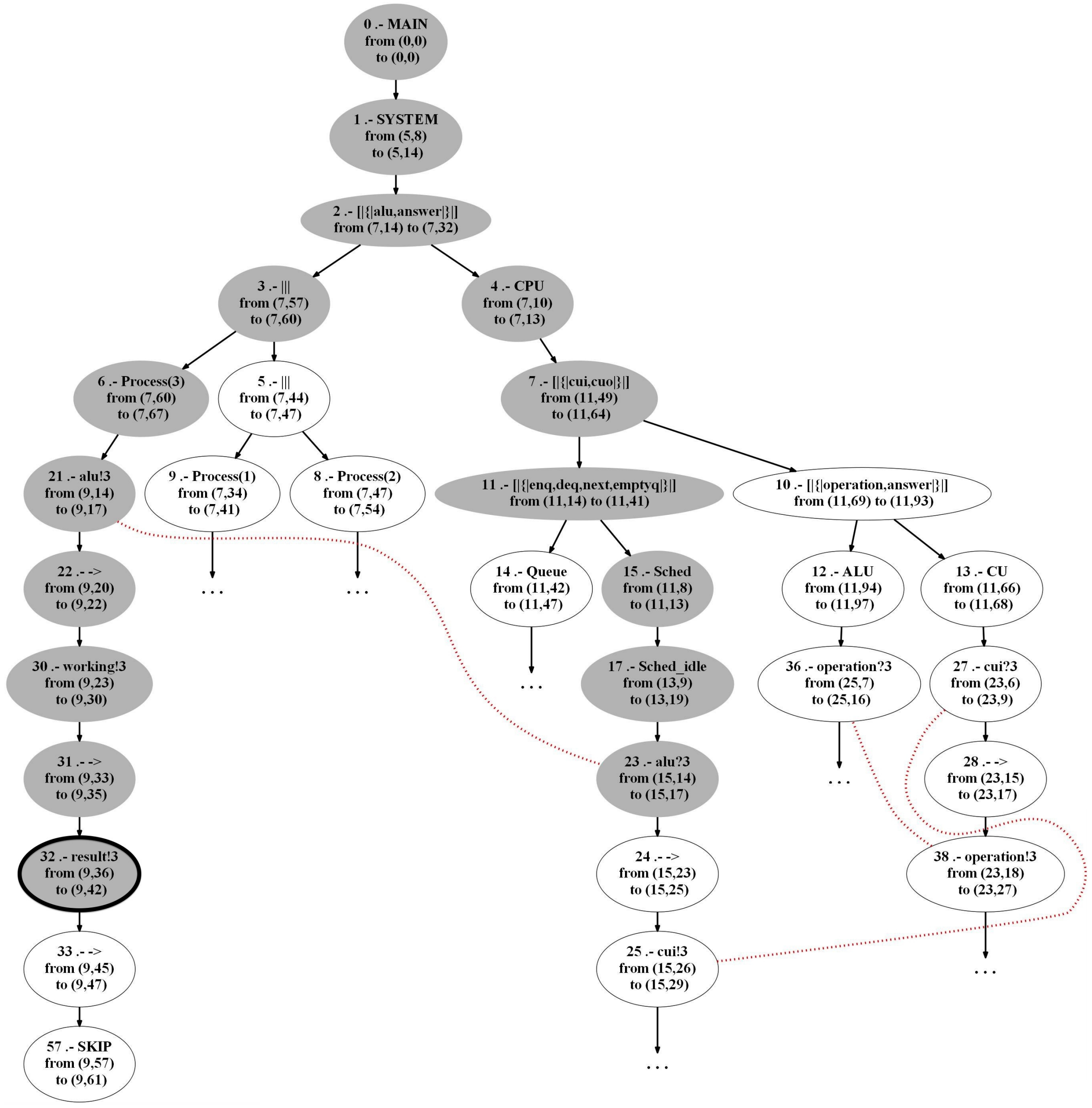

0 .- MAIN from (0,0) to (0,0)
1 .- SYSTEM from (5,8) to (5,14)
2 .- [|{|alu,answer|}|] from (7,14) to (7,32)
3 .- ||| from (7,57) to (7,60)
4 .- CPU from (7,10) to (7,13)
6 .- Process(3) from (7,60) to (7,67)
5 .- ||| from (7,44) to (7,47)
7 .- [|{|cui,cuo|}|] from (11,49) to (11,64)
21 .- alu!3 from (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 4: Partial track of the CPU specification. The node with the bold line is the slicing criterion. Dark nodes are the slice.

channels `alu` and `result` should be synchronized between the processes and the CPU. However, if we observe process `SYSTEM` in the slice, we can see that channel `result` has been accidentally replaced by `answer`. This is also obvious because process `Process` does not contain channel `answer`. Therefore, we can correct the error: $\texttt{SYSTEM} = \texttt{CPU} \underset{\{\texttt{alu,answer}\}}{||} (\texttt{Process(1)} \mid\mid\mid \texttt{Process(2)} \mid\mid\mid \texttt{Process(3)})$ should be:
$\texttt{SYSTEM} = \texttt{CPU} \underset{\{\texttt{alu,result}\}}{||} (\texttt{Process(1)} \mid\mid\mid \texttt{Process(2)} \mid\mid\mid \texttt{Process(3)})$.

This example illustrates how one can debug a CSP specification by using slices computed from tracks, or by just looking at the track, where syntactic information is shown graphically, coloured, and clearly organized with different kinds of arcs and nodes. Moreover, it shows non-explicit information in the specification, such as synchronizations.

1.1.2. Program comprehension

In Figure 2, we can observe that the track is intuitive enough as to be a powerful program comprehension tool that provides much more information than the trace. For instance, it is easy to see that process `CHECK` only communicates with process `INPUT`; and also that the execution of process `CHECK` cannot start until the other two processes have ended their communication.

In order to extend this idea and show the usefulness of tracks as a program comprehension tool, we now show an example in which we use CSP-Tracker to complement other existing tools able to (i) graphically represent all possible traces of a specification and (ii) explore traces step by step, such as FDR4[6]. CSP-Tracker is being used in the first year of the Master of Computer Science of the Universitat Politècnica de València.

This scenario is a real example that was programmed by a student learning CSP. The student was provided with a description of a system that simulates part of a casino. The student developed the CSP buggy specification (this is just a small subset) in Figure 5 (we refer the reader non-familiar with CSP to Section 3 where the CSP syntax is recalled).

```
(1)    channel betred, betblack, red, black, prize, noprize
(2)
(3)    MAIN = ROULETTE     ||      CROUPIER
                       {red,black}
(4)
(5)    ROULETTE = black → SKIP ⊓ red → SKIP
(6)
(7)    CROUPIER = betblack → black → prize → SKIP
(8)               □ betblack → red → noprize → SKIP
(9)               □ betred → red → prize → SKIP
(10)              □ betred → black → noprize → SKIP
```

Figure 5: CSP buggy specification provided by a student.

This part of the program defines a process `ROULETTE` that can behave either as a `black` or a `red` events. And a process `CROUPIER` with four different alternative cases. Both processes run in parallel and can synchronize with `black` or `red`. Even though the program is trivial, it contains a bug: when the student executed the program, it produced a deadlock. Then, the student used FDR4 to produce the graph on the left in Figure 6. This is a *static* graph that represents all possible traces. It includes the expected ones (e.g., ⟨`betred`, `red`, `prize`⟩), but it also includes four nodes that represent deadlocks (e.g., with the trace ⟨`betblack`⟩). At this point, the student knows that his program can produce a deadlock, but why the deadlock is produced remains unclear for him.

To complement the static information produced by the FDR4 graph, the student used CSP-Tracker to inspect a single computation. Fortunately, tracks include information about syntactic operators, and thus the student can observe where exactly in the program the deadlock was caused.

In the track (shown in Figure 7 left), it is clear that process `ROULETTE` deadlocked after ⊓, and process `CROUPIER` deadlocked after `betred` →. At this point, the student realized that the external choice (node 4) randomly selected the fourth branch (i.e., line 10). Hence, event `betred` was activated (node 8). On the other hand, the internal choice (node 5), randomly activated event `black`, and thus the deadlock occurred (one branch tried to synchronize with `black` and the other with `red`). The student corrected this problem replacing the internal choice by an external choice: `ROULETTE = black → SKIP ⊓ red → SKIP` should be `ROULETTE = black → SKIP □ red → SKIP`.[7]

[6] An extended description of this tool is included in Section 2.

[7] Although the solution of the student removes the deadlock, it makes the `ROULETTE` process to lose its non-deterministic nature. A better solution would be to amend `CROUPIER`:

```
ROULETTE = black → SKIP ⊓ red → SKIP
CROUPIER = betblack → (black → prize → SKIP □ red → noprize → SKIP)
           □
           betred → (red → prize → SKIP □ black → noprize → SKIP)
```

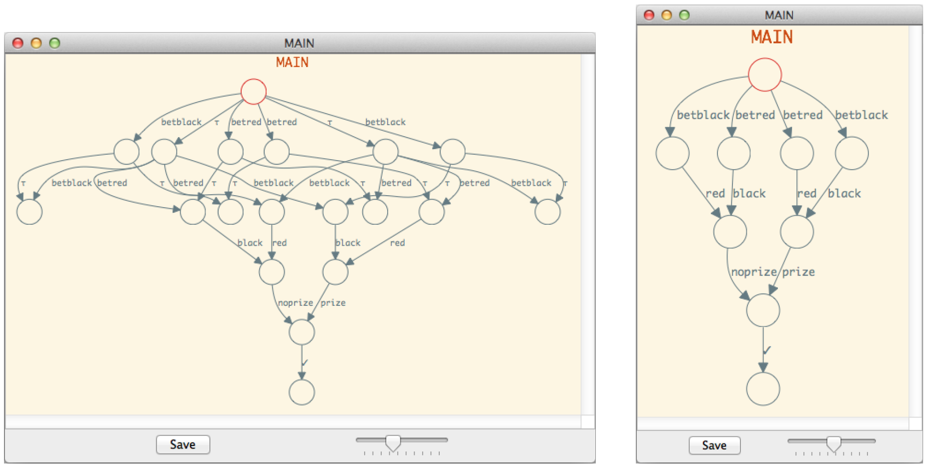


Figure 6: Graphs generated by FDR4

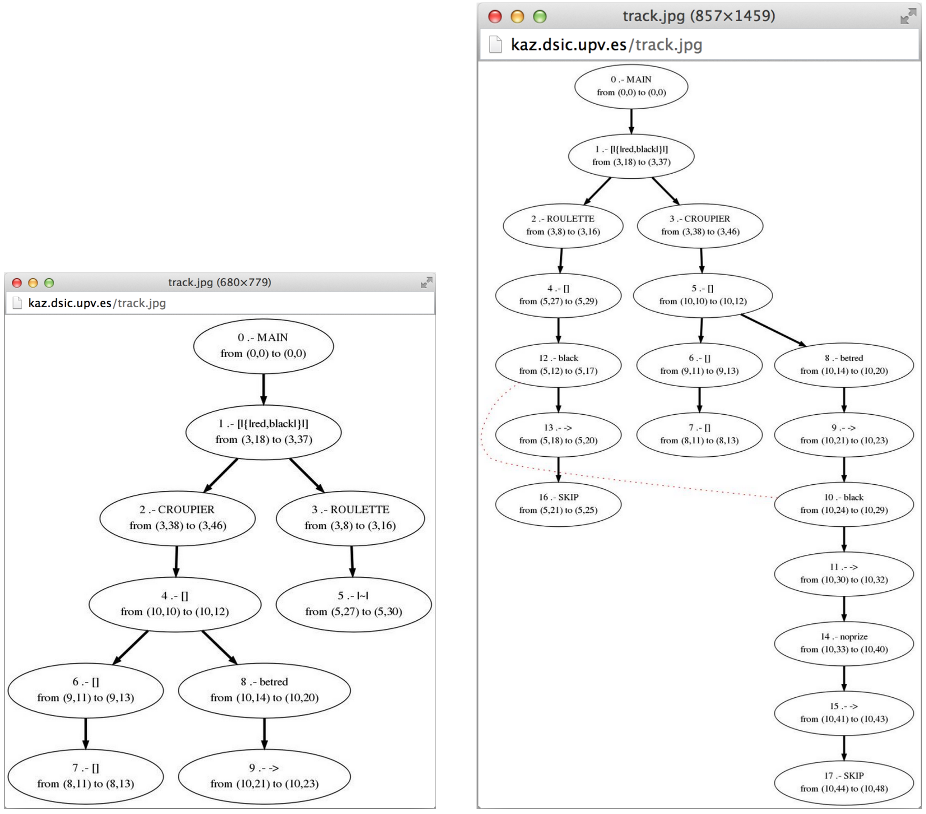

track.jpg (680×779)
kaz.dsic.upv.es/track.jpg
0 .- MAIN from (0,0) to (0,0)
1 .- [|{|red,black|}|] from (3,18) to (3,37)
2 .- CROUPIER from (3,38) to (3,46)
3 .- ROULETTE from (3,8) to (3,16)
4 .- [] from (10,10) to (10,12)
5 .- |~| from (5,27) to (5,30)
6 .- [] from (9,11) to (9,13)
8 .- betred from (10,14) to (10,20)
7 .- [] from (8,11) to (8,13)
9 .- -> from (10,21) to (10,23)
track.jpg (857×1459)
kaz.dsic.upv.es/track.jpg
0 .- MAIN from (0,0) to (0,0)
1 .- [|{|red,black|}|] from (3,18) to (3,37)
2 .- ROULETTE from (3,8) to (3,16)
3 .- CROUPIER from (3,38) to (3,46)
4 .- [] from (5,27) to (5,29)
5 .- [] from (10,10) to (10,12)
12 .- black from (5,12) to (5,17)
6 .- [] from (9,11) to (9,13)
8 .- betred from (10,14) to (10,20)
13 .- -> from (5,18) to (5,20)
7 .- [] from (8,11) to (8,13)
9 .- -> from (10,21) to (10,23)
16 .- SKIP from (5,21) to (5,25)
10 .- black from (10,24) to (10,29)
11 .- -> from (10,30) to (10,32)
14 .- noprize from (10,33) to (10,40)
15 .- -> from (10,41) to (10,43)
17 .- SKIP from (10,44) to (10,48)

Figure 7: Tracks generated by CSP-Tracker

After the program was corrected, the student generated again both the FDR4 graph (shown in Figure 6 right) and the track (shown in Figure 7 right). The syntactic information of the track is very helpful to identify the problem with precision in the program, and also to understand the different behavior of the internal and the external choices. In fact, we can observe in the track an interesting phenomenon that is not obvious for those not familiar with the semantic rules of external choice (see Figure 9): In the execution, it is possible to execute the three external choices (see nodes 4, 6 and 7 in the track on the left), even though only the first branch of the first choice was finally developed. The intuition would say that once we have executed a branch of a choice, the other branch cannot be developed. But this is not true according to the semantics (external choices can develop both branches while τ events happen), and according to the track.

1.1.3. Other applications of tracks

Another interesting application of tracks is related to component extraction and reuse. If we are interested in a particular trace, and we want to extract the part of the specification that models this trace to be used in another model, we can simply produce a slice, and slightly augment the code to make it syntactically correct (see [33, 7] for examples and explanations of this transformation, and [21, 27] for uses of dynamic slices for component extraction and reuse). For instance, consider a specification of a server with several services. In this case, we could slice the specification to obtain only the services used by a particular client and then create a specialized version of the server.

Other possible applications include clone detection, maintenance and, in general, all those analyses that need to relate the trace of a computation with its associated source code.

1.2. Contributions and structure of the paper

The goal of this paper is not to produce systems that use tracks, e.g., to perform dynamic analyses. Instead, the main goal is to formally define CSP tracks, prove properties about CSP tracks (specially their relation with CSP traces), and provide a tool to calculate them (formally, an instrumented semantics, and practically, the implementation of a CSP tracker).

In summary, the main contributions of this work are (i) the formal definition of tracks, (ii) the definition of the first tracking semantics for CSP, (iii) the proof that the trace of a computation can be extracted from the track of this computation, and (iv) the implementation of the first CSP tracker and its empirical evaluation. Concretely, we instrument the standard operational semantics of CSP in such a way that the execution of the semantics produces as a side-effect the track of the computation. It should be clear that the track of an infinite computation is also infinite. Therefore, we designed the semantics in such a way that the track is produced incrementally step by step. Thus, if the execution is stopped (e.g., by the user because it is non-terminating or due to a specified timeout), then the semantics produces the track of the computation performed so far. This semantics can serve as a theoretical foundation for tracking CSP computations because it formally relates the computations of the standard semantics with the tracks of these computations.

The rest of the paper has been organized as follows. First, in Section 2 we present and discuss the related work. Next, in Section 3 we recall the syntax and semantics of CSP. In Section 4 we define the concept of track for CSP. Then, in Section 5, we instrument the CSP semantics in such a way that its execution produces as a side-effect the track associated with the performed computation. In Section 6 we present the main results of the paper proving that the instrumented semantics presented is a conservative extension of the standard semantics, its computed tracks are correct and the corresponding trace can be extracted from the track. We describe our implementation in Section 7 (CSP-Tracker) and its empirical evaluation in Section 8. Finally, Section 9 concludes.

2. RELATED WORK

Computing CSP tracks is a complex task due to the non-deterministic execution of processes, deadlocks, non-terminating processes and mainly synchronizations. This is one reason why no correctness result exists that formally relates the track of a specification to its execution. This semantic relation is needed as it will allow us to prove important properties (such as correctness and completeness) of the techniques and tools based on tracking.

To the best of our knowledge, there does not exist any attempt to define and build tracks for CSP. Contrarily, there exists a lot of work associated with the study of CSP traces. In particular, there exist several structured trace models that provide more information than the flat list of events provided by standard traces. Two such trace models are *View-Centric Reasoning* (VCR) [50] and *structural traces* [5]. CSP and VCR tracing are both based on recording the events in a single, fairly flat trace. A VCR trace is similar to a CSP trace, but instead of recording a sequence of single events in a trace, the observer records a sequence of multisets, where each multiset represents events that can be executed in parallel. In structural tracing a structure is built up that reflects the parallel and sequential composition of the processes being traced. Traces can be composed in parallel with the || operator, and these parallel-composed traces may appear inside normal sequential traces. The following provides an example:

Example 2. *Consider the following CSP process:*

$$\mathtt{MAIN} = (\mathtt{a} \rightarrow \mathtt{b} \rightarrow \mathtt{SKIP}) \underset{\{\mathtt{a}\}}{||} (\mathtt{a} \rightarrow \mathtt{c} \rightarrow \mathtt{SKIP})$$

We have the following maximal CSP traces: $\langle \mathtt{a}, \mathtt{b}, \mathtt{c} \rangle$ and $\langle \mathtt{a}, \mathtt{c}, \mathtt{b} \rangle$. The possible maximal VCR traces are[8] *$\langle \mathtt{a}, \mathtt{b}, \mathtt{c} \rangle$, $\langle \mathtt{a}, \mathtt{c}, \mathtt{b} \rangle$ and $\langle \mathtt{a}, \{\mathtt{b}, \mathtt{c}\} \rangle$. Note that $\{\mathtt{b}, \mathtt{c}\}$ is explicitly saying that* b *and* c *could happen at the same time. In structural tracing, there is only one maximal trace: $\langle \mathtt{a}, \mathtt{b} \rangle \,||\, \langle \mathtt{a}, \mathtt{c} \rangle$. Note that a single occurrence of an event is recorded multiple times (once by each process engaging it) and when a process runs several sub-processes in parallel, information about the ordering of events between processes is lost. For instance, trace $\langle \mathtt{a}, \mathtt{b} \rangle \,||\, \langle \mathtt{a}, \mathtt{a}, \mathtt{b} \rangle$ represents a trace $\langle \mathtt{a}, \mathtt{b} \rangle$ that is in parallel with the trace $\langle \mathtt{a}, \mathtt{a}, \mathtt{b} \rangle$ (the operator || is not the usual CSP parallelism operator but a operator defined over traces instead of processes). Therefore, we cannot tell whether all the events* a *must occur before all the events* b *or any other possible pattern.*

In addition to the usual model of traces, there exist other denotational models in CSP to study the circumstances under which processes can deadlock (the *stable failures model*), and also the circumstances under which processes can livelock (the *divergences model*) [44]. All of them produce sets of traces that satisfy some property of interest.

We introduced tracks and explained its usefulness for a reduced subset of CSP in [36]. In this article we extend and complete that work by extending the syntax to a wide subset of CSP able to specify almost all industrial processes (i.e., it includes operators such as Hiding, Renaming and Conditional choice; it allows variables as parameters of processes; etc.). Moreover, this article includes the proofs of the technical results, the implementation of a CSP tracker and its empirical evaluation, and new interesting results such as a way to extract traces from tracks.

The data structure produced by FDR4's debugger is the most similar to our notion of tracks. FDR4's debugger generates counterexamples to refinement assertions that are represented with a data structure that is in many aspects related to our tracks. First, as it happens with our tracks, the debug viewer can represent several behaviours 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). The main difference between this data structure and our tracks is that 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 FDR4 are labeled 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). Arcs can be of two types: control-flow and synchronization. Control-flow arcs represent the flow of control from one node to another one. They describe that the activation of one node happened after the activation of another node. For instance, if we consider the specification $\mathtt{a} \rightarrow \mathtt{SKIP} \;\Box\; \mathtt{b} \rightarrow \mathtt{SKIP}$ and the trace $\langle \mathtt{a} \rangle$, then the track explains that prefix a was activated after choice operator $\Box$ was activated, thus the control flow passed from $\Box$ to a. This relation of the trace with the source code is missing in FDR4's graphs, and it can be useful, e.g., for program comprehension (see Section 1.1.2). In tracks, synchronizations always connect two nodes that contain two (possibly different) prefixes whose events synchronized. Hence, it explicitly says what two prefixes of the source code are being synchronized.

FDR4 also uses an LTS to represent computations using the command `:graph`. This graph is also different to a track, because it also lacks source code information, and moreover, it is a static data

[8] We write multisets of size one without set notation.

structure (it represents all possible computations) while a track is dynamic (it represents one particular computation).

Concerning the idea of confluence, introduced by Milner in [39, 40] and explored in the context of CSP by Roscoe [45, 46, 47] and used to analyse the applicability of partial-order methods in, e.g., [55], a track ensures confluence according to Lemma 10.3 in [47], because tracks are a form of truly concurrent semantics where all forms of choice other than parallel scheduling have been resolved.

Some other related work has been proposed in the context of the static analysis of CSP. In this area, some analyses are 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 (i.e., tracks are reliable—possibly infinite—representations of a computation), 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 [6] proposed a data structure based on the standard *program dependence graph* [15]. It is useful for program slicing but it is insufficient for other analyses that need a *context-sensitive graph* [26] (i.e., each different process call has a different representation). Later, a similar data structure that was context-sensitive was proposed in [33]. This structure is a graph, the *context-sensitive control flow graph*, that represents all possible computations (and maybe some unfeasible ones) of a CSP specification. In contrast, the dynamic approach presented here only represents one feasible computation. Other similar approaches, all of them static, are [8, 18].

3. THE SYNTAX AND SEMANTICS OF CSP

In order to make the paper self-contained, we recall in this section the syntax and semantics of CSP. Figure 8 summarizes the syntax constructs used in CSP specifications [44]. 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 ($\rightarrow$) It specifies that the compound object CO must happen before process P. Compound objects represent events and communications.

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

External choice ($\Box$) It is similar to internal choice but the choice comes from the external environment (e.g., the user).

Conditional choice ($\not\leq$ *Bool* $\not\geq$) 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 ($\underset{\{\overline{EV_n}\}}{||}$) 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 ($\backslash$) 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.

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)	
	$\mid$	$P \not< Bool \not> Q$	(Conditional choice)	
	$\mid$	$P \; ; \; Q$	(Sequential composition)	
	$\mid$	$P \underset{\{\overline{EV_n}\}}{\parallel} Q$	(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 8: Syntax of CSP specifications

Renaming ($[\![\Re]\!]$) 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 ($SKIP$) It is a process that successfully finishes. It allows us to continue the next sequential process if any.

Stop ($STOP$) 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., $c.a$.

We now recall the standard operational semantics of CSP as defined by A.W. Roscoe [44]. It is presented in Figure 9 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 9. 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 $subs(\overline{a_n}, \overline{x_n}, rhs(N))$ to substitute the appropriate part of $\overline{a_n}$ for each identifier in $rhs(N)$ bound by $\overline{x_n}$. This equals $rhs(N)$ if there are no identifiers bound.

(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 $co \to P$. 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.

(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.

(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.

(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)

$$\frac{}{(P \nless Bool \ngtr Q) \xrightarrow{\tau} P} \quad \text{if } Bool = true$$

(Conditional Choice 2)

$$\frac{}{(P \nless Bool \ngtr Q) \xrightarrow{\tau} Q} \quad \text{if } Bool = false$$

(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)

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

(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 9: CSP's operational semantics

(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.

We illustrate the semantics with the following example.

Example 3. *Consider the next CSP specification:*

$$\mathtt{MAIN} = (\mathtt{b} \to \mathtt{STOP})\ [\![\ \mathtt{b} \Re \mathtt{a}\ ]\!]\ \underset{\{\mathtt{a}\}}{\parallel}\ (\mathtt{P}\ \square\ (\mathtt{b} \to \mathtt{STOP}))$$
$$\mathtt{P} = (\mathtt{a} \to \mathtt{SKIP})\ ;\ \mathtt{STOP}$$

If we use MAIN *as the initial state to execute the semantics, we get the computation shown in Figure 10 where the final state is* $\mathtt{STOP}\ [\![\ \mathtt{b} \Re \mathtt{a}\ ]\!]\ \underset{\{\mathtt{a}\}}{\parallel}\ \mathtt{STOP}$*. This computation corresponds to the execution of the left branch of the choice (i.e.,* P*) and thus event* a *occurs forcing a synchronization between both processes. Each rewriting step is labeled with the applied rule, and the example should be read from top to bottom.*

(Process Call) $$\frac{}{\mathtt{MAIN} \xrightarrow{\tau} ((\mathtt{b} \to \mathtt{STOP})[\![\ \mathtt{b} \Re \mathtt{a}\ ]\!] \underset{\{\mathtt{a}\}}{\parallel} (\mathtt{P}\square(\mathtt{b} \to \mathtt{STOP})))}$$

(Synchronized Parallelism 2) $$\frac{\text{(External Choice 1)}\ \dfrac{\text{(Process Call)}\ \dfrac{}{\mathtt{P} \xrightarrow{\tau} (\mathtt{a} \to \mathtt{SKIP}); \mathtt{STOP}}}{(\mathtt{P}\square(\mathtt{b} \to \mathtt{STOP})) \xrightarrow{\tau} (((\mathtt{a} \to \mathtt{SKIP}); \mathtt{STOP})\square(\mathtt{b} \to \mathtt{STOP}))}}{(\mathtt{b} \to \mathtt{STOP})[\![\ \mathtt{b} \Re \mathtt{a}\ ]\!] \underset{\{\mathtt{a}\}}{\parallel} (\mathtt{P}\square(\mathtt{b} \to \mathtt{STOP}))) \xrightarrow{\tau} State_1}$$
$$\text{where } State_1 = (\mathtt{b} \to \mathtt{STOP})[\![\ \mathtt{b} \Re \mathtt{a}\ ]\!] \underset{\{\mathtt{a}\}}{\parallel} (((\mathtt{a} \to \mathtt{SKIP}); \mathtt{STOP})\square(\mathtt{b} \to \mathtt{STOP}))$$

(Synchronized Parallelism 3) $$\frac{Der_1 \qquad Der_2}{State_1 \xrightarrow{\mathtt{a}} (\mathtt{STOP}[\![\ \mathtt{b} \Re \mathtt{a}\ ]\!] \underset{\{\mathtt{a}\}}{\parallel} (\mathtt{SKIP}; \mathtt{STOP}))}$$

$$\text{where } Der_1 = \text{(Renaming 1)}\ \frac{\text{(Prefixing)}\ \dfrac{}{(\mathtt{b} \to \mathtt{STOP}) \xrightarrow{\mathtt{b}} \mathtt{STOP}}}{(\mathtt{b} \to \mathtt{STOP})[\![\ \mathtt{b} \Re \mathtt{a}\ ]\!] \xrightarrow{\mathtt{a}} \mathtt{STOP}[\![\ \mathtt{b} \Re \mathtt{a}\ ]\!]}$$

$$Der_2 = \text{(External Choice 3)}\ \frac{\text{(Sequential Composition 1)}\ \dfrac{\text{(Prefixing)}\ \dfrac{}{(\mathtt{a} \to \mathtt{SKIP}) \xrightarrow{\mathtt{a}} \mathtt{SKIP}}}{((\mathtt{a} \to \mathtt{SKIP}); \mathtt{STOP}) \xrightarrow{\mathtt{a}} (\mathtt{SKIP}; \mathtt{STOP})}}{(((\mathtt{a} \to \mathtt{SKIP}); \mathtt{STOP})\square(\mathtt{b} \to \mathtt{STOP})) \xrightarrow{\mathtt{a}} (\mathtt{SKIP}; \mathtt{STOP})}$$

(Synchronized Parallelism 2) $$\frac{\text{(Sequential Composition 2)}\ \dfrac{\text{(SKIP)}\ \dfrac{}{\mathtt{SKIP} \xrightarrow{\checkmark} \Omega}}{(\mathtt{SKIP}; \mathtt{STOP}) \xrightarrow{\tau} \mathtt{STOP}}}{(\mathtt{STOP}[\![\ \mathtt{b} \Re \mathtt{a}\ ]\!] \underset{\{\mathtt{a}\}}{\parallel} (\mathtt{SKIP}; \mathtt{STOP})) \xrightarrow{\tau} (\mathtt{STOP}[\![\ \mathtt{b} \Re \mathtt{a}\ ]\!] \underset{\{\mathtt{a}\}}{\parallel} \mathtt{STOP})}$$

Figure 10: A computation with the operational semantics from Fig. 9

4. TRACKING COMPUTATIONS

In this section, we provide a definition of a CSP track. Firstly, we introduce some notation that will be used throughout the paper.

A track is formed by the sequence of expressions that are evaluated during an execution. These expressions are conveniently connected to form a graph. However, several program analysis techniques such as program slicing make use of the locations of program expressions, and thus, this notion of track is insufficient for them. Therefore, we want our tracks to also store the location of each literal (i.e., events, operators and process names) in the specification so that the track can be used to know what portions of the source code have been executed and in what order. Note that this means that tracks contain dynamic and static information, i.e., the actual value of a term (with variable substitution according to the dynamic information available at execution time) is stored in the track. Additionally, the corresponding position of this term in the specification is also stored.

The inclusion of source positions in the track implies an additional level of complexity in the semantics, but the benefits of providing tracks with this additional information are clear and, for some applications, essential. Therefore, we use labels (that we call *specification positions*) to uniquely identify each literal in a specification. This roughly corresponds to nodes in the CSP specification's abstract syntax tree. We define a function $\mathcal{P}os$ to obtain the specification position of an element of a CSP specification. It is defined over nodes of the abstract syntax tree of this CSP specification. Formally,

Definition 1. *(Specification position) A* specification position *is a pair (N, w) where N is a process name, i.e., $N \in \mathcal{N}$ and w is a chain of natural numbers separated by dots $n_1.n_2.n_3...n_m$. We use Λ to denote the empty chain.*[9] *We let $\mathcal{P}os(t)$ denote the specification position of a term t. Each (parameterized) process definition $N = P$ or $N(\overline{x_n}) = P$ of a CSP specification is labeled with specification positions. The specification position of its left-hand side is respectively $\mathcal{P}os(N) = (N, 0)$ or $\mathcal{P}os(N(\overline{x_n})) = (N(\overline{x_n}), 0)$. The right-hand side is labeled with the call* $\mathtt{AddSpPos}(P, (N, \Lambda))$*; where function* `AddSpPos` *is defined as shown in Figure 11.*

$$\mathtt{AddSpPos}(P, (N, w)) = \begin{cases} P_{(N,w)} & \text{if } P \in \mathcal{N} \\ P_{(N,w)}(\overline{x_n}) & \text{if } P \in \mathcal{N} \wedge \ \overline{x_n} \in \Sigma_{\mathcal{V}} \\ STOP_{(N,w)} & \text{if } P = STOP \\ SKIP_{(N,w)} & \text{if } P = SKIP \\ co_{(N,w.1)} \rightarrow_{(N,w)} \mathtt{AddSpPos}(Q, (N, w.2)) & \text{if } P = co \rightarrow Q \\ \mathtt{AddSpPos}(Q, (N, w.1)) \backslash_{(N,w)} B & \text{if } P = Q \backslash B \\ \mathtt{AddSpPos}(Q, (N, w.1)) [\![\Re]\!]_{(N,w)} & \text{if } P = Q[\![\Re]\!] \\ \mathtt{AddSpPos}(Q, (N, w.1)) \ op_{(N,w)} \ \mathtt{AddSpPos}(R, (N, w.2)) & \text{if } P = Q \ op \ R \\ & \forall \ op \in \{\sqcap, \square, \not\prec\not\succ, ||, ;\} \end{cases}$$

Figure 11: Defintion of function `AddSpPos`

Example 4. *Consider the following CSP specification where terms are labeled with their associated specification positions (they are underlined) so that labels are unique:*

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

$$\mathtt{P(x)}_{\underline{(\mathtt{P},0)}} = (\mathtt{x}_{\underline{(\mathtt{P},1.1)}} \rightarrow_{\underline{(\mathtt{P},1)}} \mathtt{SKIP}_{\underline{(\mathtt{P},1.2)}}) \not\prec \mathtt{x} = \mathtt{c} \not\succ_{\underline{(\mathtt{P},\Lambda)}} (\mathtt{b}_{\underline{(\mathtt{P},2.1)}} \rightarrow_{\underline{(\mathtt{P},2)}} \mathtt{SKIP}_{\underline{(\mathtt{P},2.2)}})$$

All terms are uniquely labelled because labels keep the order of the associated abstract syntax tree (shown in Figure 12).

In the following, specification positions will be represented with greek letters $(\alpha, \beta, \ldots)$ and we will often use indistinguishably a term and its corresponding specification position when it is clear from the context (e.g., in Example 4 we will refer to $(\mathtt{MAIN}, 2.1)$ as `b`).

In order to introduce the formal definition of track, we need first to define the concept of *control-flow*, which refers to the order in which the terms of a CSP specification are executed. Intuitively, the control

[9] Specification positions are sequences and the empty sequence is often denoted with Λ, or also with ϵ. This notation is standard in term rewriting. See, e.g., [2] (page 36).

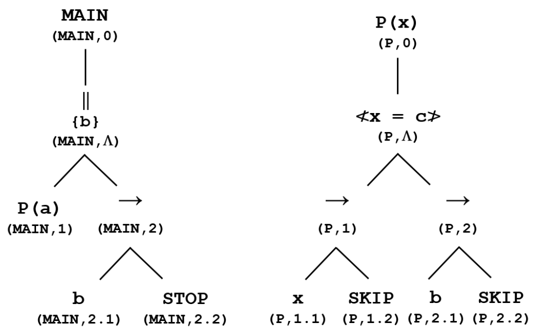


Figure 12: Abstract syntax tree

can pass from a specification position α to a specification position β iff an execution exists where α is executed before β. This notion of control-flow is similar to the control-flow used in the *control-flow graphs* (CFG) [54] of imperative programming and, thus, they are an over-approximation of the actual control-flows that can happen in an execution. We have adapted the same idea to CSP where choices and parallel composition appear; and in a similar way to the CFG, we use this definition to draw control arcs in our tracks. Formally,

Definition 2. *(Static control-flow) Given a CSP specification $\mathcal{S}$ and two specification positions α, β in $\mathcal{S}$, we say that the control* can pass from α to β*, denoted by $\alpha \Rightarrow \beta$, iff one of the following conditions holds:*

i) $\alpha = N \ \wedge \ \beta = \mathit{first}((N, \Lambda))$ with $N = rhs(N) \in \mathcal{S} \vee \quad \alpha = N(\overline{x_n}) \ \wedge \ \beta = \mathit{first}((N(\overline{x_n}), \Lambda))$ with $N(\overline{x_n}) = rhs(N(\overline{x_n})) \in \mathcal{S}$

ii) $\alpha \in \{\sqcap, \Box, \not\prec\not\succ, ||\} \ \wedge \ \beta \in \{\mathit{first}(\alpha.1), \mathit{first}(\alpha.2)\}$

iii) $\alpha \in \{\rightarrow, ;\} \ \wedge \ \beta = \mathit{first}(\alpha.2)$

iv) $\alpha = \beta.1 \ \wedge \ \beta = \rightarrow$

v) $\alpha \in \mathit{last}(\beta.1) \ \wedge \ \beta = \ ;$

vi) $\alpha \in \{\backslash, [\![\ ]\!]\} \ \wedge \ \beta = \mathit{first}(\alpha.1)$

where $\mathit{first}(\gamma)$ is the specification position of the subprocess denoted by γ which must be executed first:

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

and $\mathit{last}(\gamma)$ is the set of all possible successful termination points of the subprocess denoted by γ:

$$\mathit{last}(\gamma) = \begin{cases} \{\gamma\} & \text{if } \gamma = \texttt{SKIP} \\ \emptyset & \text{if } \gamma = \texttt{STOP} \vee \\ & \quad (\gamma = || \wedge (\mathit{last}(\gamma.1) = \emptyset \ \vee \ \mathit{last}(\gamma.2) = \emptyset)) \\ \mathit{last}(\gamma.1) \cup \mathit{last}(\gamma.2) & \text{if } \gamma \in \{\sqcap, \Box, \not\prec\not\succ\} \vee \\ & \quad (\gamma = || \wedge \mathit{last}(\gamma.1) \neq \emptyset \ \wedge \ \mathit{last}(\gamma.2) \neq \emptyset) \\ \mathit{last}(\gamma.1) & \text{if } \gamma \in \{\backslash, [\![\ ]\!]\} \\ \mathit{last}(\gamma.2) & \text{if } \gamma \in \{\rightarrow, ;\} \\ \mathit{last}((N, \Lambda)) & \text{if } \gamma = N \\ \mathit{last}((N(\overline{x_n}), \Lambda)) & \text{if } \gamma = N(\overline{x_n}) \end{cases}$$

For instance, in Example 4 we can observe how the control can pass from a specification position to another one. For instance, we have $(\texttt{P}, \Lambda) \Rightarrow (\texttt{P}, 1.1)$ and $(\texttt{P}, \Lambda) \Rightarrow (\texttt{P}, 2.1)$ due to rule ii). And $(\texttt{MAIN}, 2.1) \Rightarrow (\texttt{MAIN}, 2)$ due to rule iv); $(\texttt{MAIN}, 2) \Rightarrow (\texttt{MAIN}, 2.2)$ due to rule iii) and $(\texttt{MAIN}, 1) \Rightarrow (\texttt{P}, \Lambda)$ due to rule i).

Note that, in item iv) of Definition 2, the control can pass from the prefix to the $\rightarrow$ operator. This is a design decision because both are executed together according to the semantics, i.e., their order could be swapped. In contrast, with the ; operator (item v) of Definition 2), the order cannot be swapped because, e.g., in $P;Q$, according to the semantics, process P must finish before ; is evaluated.

We also need to define the notions of *rewriting step* and *derivation*.

Definition 3. *(Rewriting Step, Derivation) 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 contains 0, 1 or 2 rewriting steps that can be executed in any order. Given a CSP process P_0, we say that the sequence $P_0 \overset{\Theta_0}{\rightsquigarrow} \dots \overset{\Theta_n}{\rightsquigarrow} P_{n+1}$, $n \geq 0$, 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. 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.*

For instance, one (possible) complete derivation of Example 4 is:

$$\begin{array}{lll}
\texttt{MAIN} & \underset{\textsf{(PC)}}{\rightsquigarrow} & \texttt{P(a)} \underset{\{\texttt{b}\}}{\parallel} (\texttt{b} \rightarrow \texttt{STOP}) \\
& \underset{\textsf{(SP1)}}{\rightsquigarrow} & ((\texttt{a} \rightarrow \texttt{SKIP}) \not< \texttt{a} = \texttt{c} \not> (\texttt{b} \rightarrow \texttt{SKIP})) \underset{\{\texttt{b}\}}{\parallel} (\texttt{b} \rightarrow \texttt{STOP}) \\
& \underset{\textsf{(SP1)}}{\rightsquigarrow} & (\texttt{b} \rightarrow \texttt{SKIP}) \underset{\{\texttt{b}\}}{\parallel} (\texttt{b} \rightarrow \texttt{STOP}) \\
& \underset{\textsf{(SP3)}}{\rightsquigarrow} & \texttt{SKIP} \underset{\{\texttt{b}\}}{\parallel} \texttt{STOP}
\end{array}$$

where the rules applied in each rewriting step (ignoring subderivations) are (Process Call), (Synchronized Parallelism 1) and (Synchronized Parallelism 3) (abbrev. (PC), (SP1) and (SP3), respectively).

Function *last* of Definition 2 can be used to determine the last specification position in a derivation. However, this function computes all possible final specification positions, and a particular derivation only reaches some of them. Therefore, we will use in the following a modified version of *last* called *last′* whose behavior is exactly the same as *last* except in the case of choices where only one of the branches is selected. Note that, while *last* is static, *last′* is dynamic. It is defined in the context of a particular derivation which implies one particular way of resolving any non-determinism.

For each derivation $(P \sqcap Q \overset{\Theta}{\rightsquigarrow} P)$ or $(P \not< Bool \not> Q \overset{\Theta}{\rightsquigarrow} P)$ or $(P \,\Box\, Q \overset{\Theta_0}{\rightsquigarrow} \dots \overset{\Theta_n}{\rightsquigarrow} P', n \geq 0$ such that $P \overset{\Theta'_0}{\rightsquigarrow} \dots \overset{\Theta'_m}{\rightsquigarrow} P', m \geq 0)$, we have that $last'(P \sqcap Q){=}last'(P \not< Bool \not> Q){=}last'(P \,\Box\, Q){=}last'(P)$.

Similarly, for each derivation $(P \sqcap Q \overset{\Theta}{\rightsquigarrow} Q)$ or $(P \not< Bool \not> Q \overset{\Theta}{\rightsquigarrow} Q)$ or $(P \,\Box\, Q \overset{\Theta_0}{\rightsquigarrow} \dots \overset{\Theta_n}{\rightsquigarrow} Q', n \geq 0$ such that $Q \overset{\Theta'_0}{\rightsquigarrow} \dots \overset{\Theta'_m}{\rightsquigarrow} Q', m \geq 0)$, we have that $last'(P \sqcap Q){=}last'(P \not< Bool \not> Q){=}last'(P \,\Box\, Q){=}last'(Q)$.

The same happens with the definition of control-flow. Control-flow is defined statically and says whether the control can pass from α to β in some derivation. However, the track is a dynamic structure produced for a particular derivation. Therefore, we produce a dynamic version of the definition of control-flow which is defined for a particular derivation.

Definition 4. *(Dynamic control-flow) Let $\mathcal{S}$ be a CSP specification and $\mathcal{D}$ a derivation in $\mathcal{S}$. Given two specification positions α, β in $\mathcal{S}$, we say that the control* can dynamically pass from α to β, *denoted by $\alpha \Rrightarrow \beta$, iff the control can pass from α* to *β $(\alpha \Rightarrow \beta)$ in derivation $\mathcal{D}$. For each $P \overset{\Theta}{\rightsquigarrow} P' \in \mathcal{D}$ and for all rewriting steps in Θ, we have that:*

1. *if P is a prefixing $(co \to Q)$ or a sequential composition $(Q;R)$, then $\mathcal{P}os(co) \Rrightarrow \mathcal{P}os(\to)$ or $\forall p \in last'(Q)$, $\mathcal{P}os(p) \Rrightarrow \mathcal{P}os(;)$ respectively,*

2. *if $P \Rightarrow first(P'')$ where $P'' \overset{\Theta'}{\rightsquigarrow} P''' \in \Theta$, then $\mathcal{P}os(P) \Rrightarrow \mathcal{P}os(first(P''))$,*

3. *if $P \Rightarrow first(P')$, then $\mathcal{P}os(P) \Rrightarrow \mathcal{P}os(first(P'))$.*

Clauses 1, 2 and 3 define the cases in which the control passes between two specification positions in a given derivation. In clause 1, if we have a prefixing in the control then Θ is empty and the rewriting step applied is of the form $\dfrac{}{(co \to P) \xrightarrow{a} subs(a, co, P)}$ with $a \in comms(co)$. In this case, clause 1 guarantees that the control can dynamically pass from co to $\to$; and clause 3 guarantees that the control can dynamically pass from $\to$ to $subs(a, co, P)$. However, sometimes Θ is not empty, and the rewriting step is of the form $\dfrac{P'' \xrightarrow{e'} P'''}{P \xrightarrow{e} P'}$. Here, clause 2 ensures that the control can dynamically pass from P to P''; and clause 3 ensures that the control can dynamically pass from P to P' and from P'' to P'''. For instance, it is possible to have a rewriting step to evaluate process $P \,\square\, P'$. Clearly, the control can pass from $\square$ to both P and P' ($\square \Rightarrow P$ and $\square \Rightarrow P'$), but in the rewriting step the control will only pass to one of them ($\square \Rrightarrow P$ or $\square \Rrightarrow P'$). In this case, clauses 2 and 3 are used.

We are now in a position to formally define the concept of *track* of a derivation.

Definition 5. *(Track) Given a CSP specification $\mathcal{S}$, and a derivation $\mathcal{D}$ in $\mathcal{S}$, the* track *of $\mathcal{D}$ is a graph $\mathcal{G} = (N, E_c, E_s)$ where N is a set of nodes uniquely identified with a natural number and that are labeled with specification positions ($l(n)$ refers to the* label *of node n), 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 $a \mapsto a'$ iff $a \Rrightarrow a'$ with respect to $\mathcal{D}$, 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.*

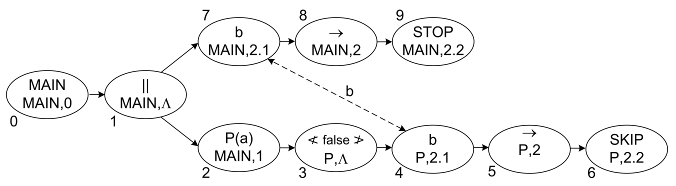
The only nodes in N are the nodes induced by E_c and E_s.

Example 5. *Consider again the specification of Example 4. Figure 13(a) shows a derivation computed with our extended semantics that will be explained in detail in the next section. The track associated with this derivation is shown in Figure 13(b). In the example, we see that the track is a connected and directed graph. Apart from the control-flow arcs, there is one synchronization arc between nodes* $(\mathtt{MAIN}, 2.1)$ *and* $(\mathtt{P}, 2.1)$ *representing the synchronization of event* $\mathtt{b}$. *To illustrate the inclusion of arcs in Definition 5, we see that the arc between node 7 and node 8 is introduced according to clause 1 of Definition 4; the arc between node 1 and node 7 is introduced according to clause 2 of Definition 4; the arc between node 3 and node 4 is introduced according to clause 3 of Definition 4 because, in the subderivation of* (SP1)*, there is a rewriting step* (Conditional Choice 2) $\dfrac{}{((\mathtt{a} \to \mathtt{SKIP}) \not\lessdot \mathtt{a} = \mathtt{c} \not\gtrdot (\mathtt{b} \to \mathtt{SKIP})) \xrightarrow{\tau} (\mathtt{b} \to \mathtt{SKIP})}$ *and $first(\mathtt{b} \to \mathtt{SKIP}) = \mathtt{b}$; similarly, the arc between nodes 5 and 6 is introduced according to clause 3 of Definition 4 because, in the subderivations of* (SP3)*, there is a rewriting step* (Prefixing) $\dfrac{}{(\mathtt{b} \to \mathtt{SKIP}) \xrightarrow{\mathtt{b}} (\mathtt{SKIP})}$ *and $first(\mathtt{SKIP}) = \mathtt{SKIP}$; and the synchronization arc between nodes 4 and 7 is introduced according to clause 2 of Definition 5.*

The trace associated with the derivation in Figure 13(a) is $\langle \mathtt{b} \rangle$*. This can be done automatically as we will prove later (see Theorem 3). Therefore, note that the track is much more informative: it shows the exact processes that have been evaluated with an explicit causality relation; and, in addition, it shows the specification positions that have been evaluated and in what order.*

$$
\begin{array}{lcl}
\mathtt{MAIN} & \underset{\text{(PC)}}{\rightsquigarrow} & \mathtt{P(a)} \underset{\{\mathtt{b}\}}{\|} (\mathtt{b} \rightarrow \mathtt{STOP}) \\
 & \underset{\text{(SP1)}}{\rightsquigarrow} & ((\mathtt{a} \rightarrow \mathtt{SKIP}) \not\lessdot \mathtt{a} = \mathtt{c} \not\gtrdot (\mathtt{b} \rightarrow \mathtt{SKIP})) \underset{\{\mathtt{b}\}}{\|} (\mathtt{b} \rightarrow \mathtt{STOP}) \\
 & \underset{\text{(SP1)}}{\rightsquigarrow} & (\mathtt{b} \rightarrow \mathtt{SKIP}) \underset{\{\mathtt{b}\}}{\|} (\mathtt{b} \rightarrow \mathtt{STOP}) \\
 & \underset{\text{(SP3)}}{\rightsquigarrow} & \mathtt{SKIP} \underset{\{\mathtt{b}\}}{\|} \mathtt{STOP} \\
 & \underline{\underset{\text{(SP1)}}{\rightsquigarrow}} & \underline{\Omega} \underline{\underset{\{\mathtt{b}\}}{\|}} \underline{\mathtt{STOP}} \\
 & \underline{\underset{\text{(SP2)}}{\rightsquigarrow}} & \underline{\Omega} \underline{\underset{\{\mathtt{b}\}}{\|}} \underline{\not\Omega}
\end{array}
$$

(a) Derivation with the extended semantics



(b) Track

Figure 13: Derivation and track associated with the specification of Example 4

5. INSTRUMENTING THE SEMANTICS FOR TRACKING

We generate tracks with an augmented semantics which is conservative with respect to the standard operational semantics. Therefore, the evaluation of the semantic rules follows the standard order producing the standard trace. Moreover, the semantics generates the track incrementally, step by step. Therefore, infinite computations can be tracked until they are stopped. Hence, it is not needed to actually finish a computation to get the track of the subcomputations performed.

Example 6. *In the following CSP specification two non-terminating processes run in parallel and synchronize infinitely.*

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

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

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

Because the computation is infinite, the track (shown in Figure 14) is also infinite.

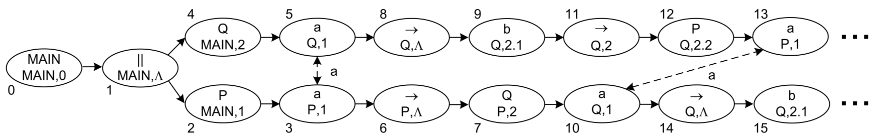


Figure 14: Track of the program in Example 6

In the standard semantics there does not exist a rule for process `STOP`, and thus it produces a deadlock. Contrarily, the augmented semantics includes a special rule to handle `STOP`. This rule allows us to perform one additional step to generate the part of the track that represents this deadlock. This additional step does not influence the other rules of the semantics, thus it remains conservative.

This section introduces an instrumented operational semantics of CSP which generates as a side-effect the tracks associated with the computations performed with the semantics. The tracking semantics is shown in Figure 15 and Figure 16, where we assume that every term in the program has been labeled with its specification position (denoted by a subscript, e.g., P_α). In this semantics, a *state* is a tuple $(P, \mathcal{G}, m)$, where P is the process to be evaluated (the *control*), $\mathcal{G}$ is a directed graph (i.e., the track built so far) and m is a numeric reference to the current node in $\mathcal{G}$. Concretely, m references the node in $\mathcal{G}$ where the specification position of the control P must be stored. Reference m is a fresh[10] reference generated to add new nodes to $\mathcal{G}$. The basic idea of the graph construction is to record the current control with the current reference in every step by connecting it to its parent. We use the notation $\mathcal{G}[m \underset{n}{\mapsto} (\alpha, t)]$ to introduce in $\mathcal{G}$ a node labeled with term t and its specification position α; with reference m and with successor n (a fresh reference). t is part of the expression that is being evaluated, and it usually contains variables that have been substituted by their corresponding values. For the sake of simplicity, whenever t does not contain variables, we just omit t because it can be directly inferred from the specification position. Hence, in these cases we just use $\mathcal{G}[m \underset{n}{\mapsto} \alpha]$. Successor arrows are denoted by $m \underset{n}{\mapsto}$ which means that node n is the successor of node m.

Together with the events that fire the rules of the semantics, there is an associated set Δ. This set contains references to nodes that may be synchronized. In particular, every time an event in Σ happens during the computation, its reference is stored in the set Δ of the current state. Therefore, when a synchronized parallelism is evaluated, all the events that must be synchronized are in Δ. We use the special symbol $\not\Omega$ to denote any process that is deadlocked. In order to perform computations, we construct an initial state (e.g., $(\mathtt{MAIN}_{(\mathtt{MAIN},0)}, \emptyset, 0)$) and (non-deterministically) apply the rules of Figure 15 and Figure 16. When the execution has finished or has been interrupted, the semantics has produced the track of the computation performed so far.

An explanation for each rule of the semantics follows:

((Parameterized) Process Call) The called process N is unfolded, node m is added to the graph with specification position α and successor n (a fresh reference). The new process in the control is $rhs(N)$. The set Δ of events to be synchronized is put to $\emptyset$ since no event in Σ has been fired. Parameterized process call is completely analogous, but in this case (as in the standard semantics) we need to replace all variables appearing in the right hand side of the process ($rhs(N)$) by their corresponding values which are the parameters $\overline{a_n}$ of N.

(Prefixing) This rule adds nodes m (the prefix) and n (the prefixing operator) to the graph. In the new state, n becomes the parent reference and the fresh reference p represents the current reference. The new control is P with the corresponding substitution as in the standard semantics. The set Δ is $\{m\}$ to record the node associated to event a that must be synchronized when required by (Synchronized Parallelism 3).

(SKIP and STOP) Whenever one of these rules is applied, the subcomputation finishes because Ω (for rule `SKIP`) and $\not\Omega$ (for rule `STOP`) are put in the control, and these special symbols have no associated rule. A node with the `SKIP` (respectively `STOP`) specification position is added to the graph.

(Internal Choice 1 and 2) The choice operator is added to the graph, and the (non-deterministically) selected branch is put into the control with the fresh reference n as the successor of the choice operator.

(External Choice 1, 2, 3 and 4) An external choice is able to develop both branches while τ events happen (rules 1 and 2), until an event in $\Sigma \cup \{\checkmark\}$ occurs (rules 3 and 4). This means that the semantics can

[10]We assume that fresh references are numeric and generated incrementally.

(Process Call)

$$\frac{}{(N_\alpha, \mathcal{G}, m) \xrightarrow{(\tau,\emptyset)} (rhs(N), \mathcal{G}[m \underset{n}{\mapsto} \alpha], n)}$$

(Parameterized Process Call)

$$\frac{}{(N_\alpha(\overline{a_n}), \mathcal{G}, m) \xrightarrow{(\tau,\emptyset)} (subs(\overline{a_n}, \overline{x_n}, rhs(N)), \mathcal{G}[m \underset{n}{\mapsto} (\alpha, N(\overline{a_n}))], n)}$$
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)

$$\frac{}{(co_\alpha \rightarrow_\beta P, \mathcal{G}, m) \xrightarrow{(a,\{m\})} (subs(a, co, P), \mathcal{G}', p)} \quad a \in comms(co)$$
where $\mathcal{G}' = \mathcal{G}[m \underset{n}{\mapsto} (\alpha, subs(a, co, co)), n \underset{p}{\mapsto} \beta]$

(SKIP)

$$\frac{}{(\texttt{SKIP}_\alpha, \mathcal{G}, m) \xrightarrow{(\checkmark,\emptyset)} (\Omega, \mathcal{G}[m \underset{n}{\mapsto} \alpha], n)}$$

(STOP)

$$\frac{}{(\texttt{STOP}_\alpha, \mathcal{G}, m) \xrightarrow{(\tau,\emptyset)} (\not\Omega, \mathcal{G}[m \underset{n}{\mapsto} \alpha], n)}$$

(Internal Choice 1)

$$\frac{}{(P \sqcap_\alpha Q, \mathcal{G}, m) \xrightarrow{(\tau,\emptyset)} (P, \mathcal{G}[m \underset{n}{\mapsto} \alpha], n)}$$

(Internal Choice 2)

$$\frac{}{(P \sqcap_\alpha Q, \mathcal{G}, m) \xrightarrow{(\tau,\emptyset)} (Q, \mathcal{G}[m \underset{n}{\mapsto} \alpha], n)}$$

(External Choice 1)

$$\frac{(P_1, \mathcal{G}', n') \xrightarrow{(\tau,\emptyset)} (P', \mathcal{G}'', n'')}{(P_1 \,\square_{(\alpha,n_1,n_2)} P_2, \mathcal{G}, m) \xrightarrow{(\tau,\emptyset)} (P' \,\square_{(\alpha,n'',n_2)} P_2, \mathcal{G}'', m)}$$
where $(\mathcal{G}', n') \ = \ \texttt{FirstEval}(\mathcal{G}, n_1, m, (\alpha, \square))$

(External Choice 2)

$$\frac{(P_2, \mathcal{G}', n') \xrightarrow{(\tau,\emptyset)} (P', \mathcal{G}'', n'')}{(P_1 \,\square_{(\alpha,n_1,n_2)} P_2, \mathcal{G}, m) \xrightarrow{(\tau,\emptyset)} (P_1 \,\square_{(\alpha,n_1,n'')} P', \mathcal{G}'', m)}$$
where $(\mathcal{G}', n') \ = \ \texttt{FirstEval}(\mathcal{G}, n_2, m, (\alpha, \square))$

(External Choice 3)

$$\frac{(P_1, \mathcal{G}', n') \xrightarrow{(e,\Delta)} (P', \mathcal{G}'', n'')}{(P_1 \,\square_{(\alpha,n_1,n_2)} P_2, \mathcal{G}, m) \xrightarrow{(e,\Delta)} (P', \mathcal{G}'', n'')} \quad e \in \Sigma \cup \{\checkmark\}$$
where $(\mathcal{G}', n') \ = \ \texttt{FirstEval}(\mathcal{G}, n_1, m, (\alpha, \square))$

(External Choice 4)

$$\frac{(P_2, \mathcal{G}', n') \xrightarrow{(e,\Delta)} (P', \mathcal{G}'', n'')}{(P_1 \,\square_{(\alpha,n_1,n_2)} P_2, \mathcal{G}, m) \xrightarrow{(e,\Delta)} (P', \mathcal{G}'', n'')} \quad e \in \Sigma \cup \{\checkmark\}$$
where $(\mathcal{G}', n') \ = \ \texttt{FirstEval}(\mathcal{G}, n_2, m, (\alpha, \square))$

(Conditional Choice 1)

$$\frac{}{(P \not< Bool \not>_\alpha Q, \mathcal{G}, m) \xrightarrow{(\tau,\emptyset)} (P, \mathcal{G}[m \underset{n}{\mapsto} (\alpha, \not< true \not>)], n)} \quad \text{if } Bool = true$$

(Conditional Choice 2)

$$\frac{}{(P \not< Bool \not>_\alpha Q, \mathcal{G}, m) \xrightarrow{(\tau,\emptyset)} (Q, \mathcal{G}[m \underset{n}{\mapsto} (\alpha, \not< false \not>)], n)} \quad \text{if } Bool = false$$

Figure 15: An instrumented operational semantics to generate CSP tracks

add nodes to both branches of the track alternatively, and thus, it needs to store the next reference to use in every branch of the choice. This is done by labeling choice operators with a tuple of the form (α, n_1, n_2) where α is the specification position of the choice operator; and n_1 and n_2 are respectively the references to be used in the left and right branches of the choice, and they are initialized to $\bullet$, a symbol used to express that the branch has not been evaluated yet. Therefore, the first time a branch is evaluated, we generate a new reference for this branch. For this purpose, function `FirstEval` is used:

$$\texttt{FirstEval}(\mathcal{G}, n, m, (\alpha, t)) \ = \ \begin{cases} (\mathcal{G}[m \underset{p}{\mapsto} (\alpha, t)], p) & \text{if } n = \bullet \\ (\mathcal{G}, n) & \text{otherwise} \end{cases}$$

(Sequential Composition 1)

$$\frac{(P, \mathcal{G}, m) \xrightarrow{(e,\Delta)} (P', \mathcal{G}', m')}{(P; Q, \mathcal{G}, m) \xrightarrow{(e,\Delta)} (P'; Q, \mathcal{G}', m')} \quad e \in \Sigma \cup \{\tau\}$$

(Sequential Composition 2)

$$\frac{(P, \mathcal{G}, m) \xrightarrow{(\checkmark,\emptyset)} (\Omega, \mathcal{G}', n)}{(P;_\alpha Q, \mathcal{G}, m) \xrightarrow{(\tau,\emptyset)} (Q, \mathcal{G}'[n \underset{p}{\mapsto} \alpha], p)}$$

(Synchronized Parallelism 1)

$$\frac{(P_1, \mathcal{G}', n') \xrightarrow{(e',\Delta)} (P', \mathcal{G}'', n'')}{(P_1 \underset{X}{\|}_{(\alpha,n_1,n_2)} P_2, \mathcal{G}, m) \xrightarrow{(e,\Delta)} (P' \underset{X}{\|}_{(\alpha,n'',n_2)} P_2, \mathcal{G}'', m)} \quad \begin{array}{l}(e = e' \in \Sigma \backslash X)\\ \vee\ (e = \tau\ \wedge\ e' \in \{\tau, \checkmark\})\end{array}$$

where $(\mathcal{G}', n') \ = \ \texttt{FirstEval}(\mathcal{G}, n_1, m, (\alpha, \underset{X}{\|}))$

(Synchronized Parallelism 2)

$$\frac{(P_2, \mathcal{G}', n') \xrightarrow{(e',\Delta)} (P', \mathcal{G}'', n'')}{(P_1 \underset{X}{\|}_{(\alpha,n_1,n_2)} P_2, \mathcal{G}, m) \xrightarrow{(e,\Delta)} (P_1 \underset{X}{\|}_{(\alpha,n_1,n'')} P', \mathcal{G}'', m)} \quad \begin{array}{l}(e = e' \in \Sigma \backslash X)\\ \vee\ (e = \tau\ \wedge\ e' \in \{\tau, \checkmark\})\end{array}$$

where $(\mathcal{G}', n') \ = \ \texttt{FirstEval}(\mathcal{G}, n_2, m, (\alpha, \underset{X}{\|}))$

(Synchronized Parallelism 3)

$$\frac{\mathit{RewritingStep}_1 \qquad\qquad \mathit{RewritingStep}_2}{(P_1 \underset{X}{\|}_{(\alpha,n_1,n_2)} P_2, \mathcal{G}, m) \xrightarrow{(a,\Delta_1 \cup \Delta_2)} (P'_1 \underset{X}{\|}_{(\alpha,n''_1,n''_2)} P'_2, \mathcal{G}'', m)} \quad a \in X$$

where $\mathcal{G}'' = \mathcal{G}''_1 \cup \mathcal{G}''_2 \cup \{s_1 \overset{a}{\leftrightarrow} s_2 \mid s_1 \in \Delta_1 \wedge s_2 \in \Delta_2\}$

$\wedge\ \mathit{RewritingStep}_1 = (P_1, \mathcal{G}'_1, n'_1) \xrightarrow{(a,\Delta_1)} (P'_1, \mathcal{G}''_1, n''_1)$

$\wedge\ (\mathcal{G}'_1, n'_1) \ = \ \texttt{FirstEval}(\mathcal{G}, n_1, m, (\alpha, \underset{X}{\|}))$

$\wedge\ \mathit{RewritingStep}_2 = (P_2, \mathcal{G}'_2, n'_2) \xrightarrow{(a,\Delta_2)} (P'_2, \mathcal{G}''_2, n''_2)$

$\wedge\ (\mathcal{G}'_2, n'_2) \ = \ \texttt{FirstEval}(\mathcal{G}, n_2, m, (\alpha, \underset{X}{\|}))$

(Synchronized Parallelism 4)

$$\frac{}{(\Omega \underset{X}{\|}_{(\alpha,n_1,n_2)} \Omega, \mathcal{G}, m) \xrightarrow{(\checkmark,\emptyset)} (\Omega, \mathcal{G}', r)}$$

where $\mathcal{G}' = \mathcal{G}[\{p \underset{r}{\mapsto} \mid p \underset{q}{\mapsto} \ \in \mathcal{G} \text{ where } q \in \{n_1, n_2\}\}]$

(Hiding 1)

$$\frac{(P, \mathcal{G}', n) \xrightarrow{(a,\Delta)} (P', \mathcal{G}'', n')}{(P \backslash_\alpha B, \mathcal{G}, m) \xrightarrow{(\tau,\emptyset)} (P' \backslash_\bullet B, \mathcal{G}'', n')} \quad a \in B$$

(Hiding 2)

$$\frac{(P, \mathcal{G}', n) \xrightarrow{(e,\Delta)} (P', \mathcal{G}'', n')}{(P \backslash_\alpha B, \mathcal{G}, m) \xrightarrow{(e,\Delta)} (P' \backslash_\bullet B, \mathcal{G}'', n')} \quad (e \in \Sigma \wedge e \notin B)\ \vee\ (e = \tau)$$

(Hiding 3)

$$\frac{(P, \mathcal{G}', n) \xrightarrow{(\checkmark,\emptyset)} (\Omega, \mathcal{G}'', n')}{(P \backslash_\alpha B, \mathcal{G}, m) \xrightarrow{(\checkmark,\emptyset)} (\Omega, \mathcal{G}'', n')}$$

$(\mathcal{G}', n) \ = \ \texttt{FirstEval}_{\texttt{HR}}(\mathcal{G}, (\alpha, \backslash B), m)$

(Renaming 1)

$$\frac{(P, \mathcal{G}', n) \xrightarrow{(e',\Delta)} (P', \mathcal{G}'', n')}{(P[\![\Re]\!]_\alpha, \mathcal{G}, m) \xrightarrow{(e,\Delta)} (P'[\![\Re]\!]_\bullet, \mathcal{G}'', n')} \quad \begin{array}{l}(e, e' \in \Sigma \wedge e'\ \Re\ e)\ \vee\\ \quad (e = e' = \tau)\end{array}$$

(Renaming 2)

$$\frac{(P, \mathcal{G}', n) \xrightarrow{(\checkmark,\emptyset)} (\Omega, \mathcal{G}'', n')}{(P[\![\Re]\!]_\alpha, \mathcal{G}, m) \xrightarrow{(\checkmark,\emptyset)} (\Omega, \mathcal{G}'', n')}$$

$(\mathcal{G}', n) \ = \ \texttt{FirstEval}_{\texttt{HR}}(\mathcal{G}, (\alpha, [\![\Re]\!]), m)$

Figure 16: An instrumented operational semantics to generate CSP tracks (cont.)

This function checks whether this is the first time that the branch is evaluated (this only happens when the reference of this branch is empty, i.e., $n = \bullet$). In this case, the choice operator is added to $\mathcal{G}$. For instance, consider the following CSP specification:

$\texttt{MAIN} = \texttt{P} \;\Box\; \texttt{SKIP}$

$\texttt{P} = \texttt{SKIP}$

A call to process `MAIN` will leave $\texttt{P} \;\Box\; \texttt{SKIP}$ in the control of a rewriting step (External Choice 1) like the following:

$$\text{(EC1)}\;\dfrac{\text{(Process Call)}\;\dfrac{}{(\texttt{P}, \mathcal{G}', 2) \xrightarrow{(\tau,\emptyset)} (\texttt{SKIP}, \mathcal{G}'' = \mathcal{G}'[2 \underset{3}{\mapsto} (\texttt{MAIN}, 1)], 3)}}{(\texttt{P} \;\Box_{((\texttt{MAIN},\Lambda),\bullet,\bullet)}\; \texttt{SKIP}, \mathcal{G}, 1) \xrightarrow{(\tau,\emptyset)} (\texttt{SKIP} \;\Box_{((\texttt{MAIN},\Lambda),3,\bullet)}\; \texttt{SKIP}, \mathcal{G}'', 1)}$$

where $\texttt{FirstEval}(\mathcal{G}, \bullet, 1, ((\texttt{MAIN}, \Lambda), \Box)) \;=\; (\mathcal{G}' = \mathcal{G}[1 \underset{2}{\mapsto} ((\texttt{MAIN}, \Lambda), \Box)], 2)$.

Observe that, in the left control, the choice operator is labeled with $((\texttt{MAIN}, \Lambda), \bullet, \bullet)$. Therefore, it is evaluated for the first time, and thus, node $1 \underset{2}{\mapsto} ((\texttt{MAIN}, \Lambda), \Box)$ is added to $\mathcal{G}$ by `FirstEval`. Note that 1 refers to the choice operator and 2 refers to the first node of the left branch (`P`). Note also that, in the right control, the left $\bullet$ is replaced by the corresponding reference 3. We continue the evaluation with the following rewriting step (External Choice 3):

$$\text{(EC3)}\;\dfrac{\text{(SKIP)}\;\dfrac{}{(\texttt{SKIP}, \mathcal{G}'', 3) \xrightarrow{(\checkmark,\emptyset)} (\Omega, \mathcal{G}''' = \mathcal{G}''[3 \underset{4}{\mapsto} (\texttt{MAIN}, 1)], 4)}}{(\texttt{SKIP} \;\Box_{((\texttt{MAIN},\Lambda),3,\bullet)}\; \texttt{SKIP}, \mathcal{G}'', 1) \xrightarrow{(\checkmark,\emptyset)} (\Omega, \mathcal{G}'', 4)}$$

where $\texttt{FirstEval}(\mathcal{G}'', 3, 3, ((\texttt{MAIN}, \Lambda), \Box)) \;= (\mathcal{G}'', 3)$.

In this case, `FirstEval` does not modify the graph ($\mathcal{G}''$) because the left branch was already evaluated. This is known because the left reference of the choice operator is not a $\bullet$.

(Conditional Choice 1 and 2) It is completely analogous to (Internal Choice) but the selection of the branch is deterministic and comes from the evaluation of the condition.

(Sequential Composition 1 and 2) The first rule is used to evolve process P until it is finished. P is evolved to P' which is put into the control. When P successfully finishes (it becomes Ω), $\checkmark$ happens. Then, (Sequential Composition 2) is used and Q is put into the control. The sequential composition operator ; is added to the graph with successor p that is the reference to be used in the first node added in the subderivation associated with Q.

(Synchronized Parallelism 1 and 2) In a synchronized parallel composition, both parallel processes can be intertwiningly executed until a synchronized event is found. Therefore, nodes from both processes can be added interwoven to the graph. Hence, each parallelism operator is labeled with a tuple of the form (α, n_1, n_2) as it happens with external choices.

These rules develop the branches of the parallelism until they are finished or until they must synchronize. In order to introduce the parallelism operator into the graph, function `FirstEval` is used, as in the external choice rules. For instance, consider the first rewriting step (Synchronized Parallelism 1) of Figure 17. The parallelism operator in the rewriting step *State 1* is labeled with $((\texttt{MAIN}, \Lambda), \bullet, \bullet)$. Therefore, it is evaluated for the first time, and thus, in the left-hand side state of the rewriting step (Parameterized Process Call), node $1 \underset{2}{\mapsto} (\texttt{MAIN}, \Lambda)$ is added to $\mathcal{G}$. Note that 1 refers to the parallelism operator and 2 refers to `P(a)`, which is the first position in the left branch.

(Synchronized Parallelism 3) This rule is used to synchronize the parallel processes. In this case, both branches must perform a rewriting step with the same visible event. Each branch derivation has a non-empty set of references to events (Δ_1, Δ_2) to be synchronized (note that this is a set because events belong to (possibly many nested) parallelisms). Observe that synchronized events can be pairwise different due to the occurrence of renaming. All references in the sets Δ_1 and Δ_2 are mutually linked with synchronization arcs ($\overset{a}{\leftrightarrow}$). Both sets are finally joined to form the new set of synchronized events.

(Synchronized Parallelism 4) It is used when none of the parallel processes can proceed because they already successfully finished. In this case, the control becomes Ω indicating the successful termination of the synchronized parallelism. In the new state, the new (fresh) reference is r. This rule also adds to the graph the arcs from all the parents of the last references of each branch (n_1 and n_2) to r. Here, we use the notation $p \underset{r}{\mapsto}$ to add an arc from p to r. Note that the fact of generating the next reference in each rule allows (Synchronized Parallelism 4) to connect the final node of both branches to the next node. This simplifies other rules such as (Sequential Composition) that already has the reference of the node ready.

(Hiding 1, 2 and 3) (Hiding 1) is used to hide an event in P that belongs to the hiding set B. Events are hidden with τ. Because the event is hidden, it should not be synchronized by other rules; therefore, the set Δ is put to $\emptyset$. If the event does not belong to B then (Hiding 2) is used, and thus, it remains observable from outside. In both cases, the specification position of the hiding operator is replaced by $\bullet$ (in the next state of the semantics, not in the specification) meaning that it has been already evaluated. This is used to ensure that the hiding operator is only added to the graph once. For this purpose, function $\mathtt{FirstEval_{HR}}$ is used:

$$\mathtt{FirstEval_{HR}}(\mathcal{G}, (\alpha, t), m) = \begin{cases} (\mathcal{G}[m \underset{n}{\mapsto} (\alpha, t)], n) & \text{if } \alpha \neq \bullet \\ (\mathcal{G}, m) & \text{otherwise} \end{cases}$$

Function $\mathtt{FirstEval_{HR}}$ checks the specification position of the hiding operator. If it is not $\bullet$, then it is the first time that it is evaluated and thus it is added to the graph. (Hiding 3) is used to finish the process by placing Ω in the control and performing $\checkmark$.

(Renaming 1 and 2) It is completely analogous to the previous case, but here the event is not hidden, but replaced by another event in mapping $\Re$. Note in (Renaming 1) that, contrarily to (Hiding 1), the set Δ is passed down from the top rewriting step. This is done because an event a happened that has been added to Δ and it must be synchronized if required by another rule. Due to the renaming, other rules see a as b, so, if they try to synchronize b, they will use the reference of a included in Δ.

We illustrate this semantics with a simple example.

Example 7. *Consider again the specification in Example 4. Figure 13(a) shows one possible derivation (excluding subderivations) for this example. Note that the underlined part corresponds to the additional rewriting steps performed by the tracking semantics. This derivation corresponds to the execution of the instrumented semantics with the initial state* $(\mathtt{MAIN}_{(\mathtt{MAIN},0)}, \emptyset, 0)$ *shown in Figure 17. Here, for clarity, each computation step is labeled with the applied rule; in each state, $\mathcal{G}$ denotes the current graph. This computation corresponds to the execution of the right branch of the conditional choice (i.e.,* $\mathtt{b} \rightarrow \mathtt{SKIP}$*). The final state is* $(\Omega \underset{\{\mathtt{b}\}}{\parallel}_{((\mathtt{MAIN},\Lambda),10,11)} \not\Omega, \mathcal{G}', 1)$. *The final track $\mathcal{G}'$ computed for this execution is depicted in Figure 13(b) where we can see that nodes are numbered with the references generated by the instrumented semantics. Note that nodes 10 and 11 were prepared by the semantics (arcs to them were produced) but never used because the subcomputations were stopped in* $\mathtt{STOP}$. *Note also that the track contains all the parts of the specification executed by the semantics.*

$$\text{(Process Call)}\,\dfrac{}{(\mathtt{MAIN},\emptyset,0)\xrightarrow{(\tau,\emptyset)}\mathit{State}\ 1}\ \text{where}$$

$$\mathit{State}\ 1 = \ (\mathtt{P(a)}\ \underset{\{\mathtt{b}\}}{\|}_{((\mathtt{MAIN},\Lambda),\bullet,\bullet)}(\mathtt{b}\rightarrow\mathtt{STOP}),\mathcal{G}[0\underset{1}{\mapsto}(\mathtt{MAIN},0)],1)$$

$$\text{(SP 1)}\,\dfrac{\text{(PPC)}\,\dfrac{}{(\mathtt{P(a)},\mathcal{G}[1\underset{2}{\mapsto}(\mathtt{MAIN},\Lambda)],2)\xrightarrow{(\tau,\emptyset)}((\mathtt{a}\rightarrow\mathtt{SKIP})\not<\mathtt{a}=\mathtt{c}\not>(\mathtt{b}\rightarrow\mathtt{SKIP}),\mathcal{G}[2\underset{3}{\mapsto}((\mathtt{MAIN},1),\mathtt{P(a)})],3)}}{\mathit{State}\ 1\xrightarrow{(\tau,\emptyset)}\mathit{State}\ 2}$$

$$\text{where } \mathit{State}\ 2 = (((\mathtt{a}\rightarrow\mathtt{SKIP})\not<\mathtt{a}=\mathtt{c}\not>(\mathtt{b}\rightarrow\mathtt{SKIP}))\ \underset{\{\mathtt{b}\}}{\|}_{((\mathtt{MAIN},\Lambda),3,\bullet)}(\mathtt{b}\rightarrow\mathtt{STOP}),\mathcal{G}',1)$$

$$\text{(SP 1)}\,\dfrac{\text{(CC 2)}\,\dfrac{}{((\mathtt{a}\rightarrow\mathtt{SKIP})\not<\mathtt{a}=\mathtt{c}\not>(\mathtt{b}\rightarrow\mathtt{SKIP}),\mathcal{G},3)\xrightarrow{(\tau,\emptyset)}(\mathtt{b}\rightarrow\mathtt{SKIP},\mathcal{G}[3\underset{4}{\mapsto}((\mathtt{P},\Lambda),\not<\mathtt{false}\not>)],4)}}{\mathit{State}\ 2\xrightarrow{(\tau,\emptyset)}\mathit{State}\ 3}$$

$$\text{where } \mathit{State}\ 3 = ((\mathtt{b}\rightarrow\mathtt{SKIP})\ \underset{\{\mathtt{b}\}}{\|}_{((\mathtt{MAIN},\Lambda),4,\bullet)}(\mathtt{b}\rightarrow\mathtt{STOP}),\mathcal{G}',1)$$

$$\text{(SP 3)}\,\dfrac{L\quad R}{\mathit{State}\ 3\xrightarrow{(\mathtt{b},\{4,7\})}\mathit{State}\ 4}\ \text{where}$$

$$L = \text{(Prefixing)}\,\dfrac{}{(\mathtt{b}\rightarrow\mathtt{SKIP},\mathcal{G},4)\xrightarrow{(\mathtt{b},\{4\})}(\mathtt{SKIP},\mathcal{G}[4\underset{5}{\mapsto}(\mathtt{P},2.1),5\underset{6}{\mapsto}(\mathtt{P},2)],6)}$$

$$R = \text{(Prefixing)}\,\dfrac{}{(\mathtt{b}\rightarrow\mathtt{STOP},\mathcal{G}[1\underset{7}{\mapsto}(\mathtt{MAIN},\Lambda)],7)\xrightarrow{(\mathtt{b},\{7\})}(\mathtt{STOP},\mathcal{G}[7\underset{8}{\mapsto}(\mathtt{MAIN},2.1),8\underset{9}{\mapsto}(\mathtt{MAIN},2)],9)}$$

$$\text{and } \mathit{State}\ 4 = \ (\mathtt{SKIP}\ \underset{\{\mathtt{b}\}}{\|}_{((\mathtt{MAIN},\Lambda),6,9)}\mathtt{STOP},\mathcal{G}'\cup\{4\overset{\mathtt{b}}{\leftrightarrow}7\},1)$$

$$\text{(SP 1)}\,\dfrac{\text{(SKIP)}\,\dfrac{}{(\mathtt{SKIP},\mathcal{G},6)\xrightarrow{(\tau,\emptyset)}(\Omega,\mathcal{G}[6\underset{10}{\mapsto}(\mathtt{P},2.2)],10)}}{\mathit{State}\ 4\xrightarrow{(\tau,\emptyset)}\mathit{State}\ 5}$$

$$\text{where } \mathit{State}\ 5 = (\Omega\ \underset{\{\mathtt{b}\}}{\|}_{((\mathtt{MAIN},\Lambda),10,9)}\mathtt{STOP},\mathcal{G}',1)$$

$$\text{(SP 2)}\,\dfrac{\text{(STOP)}\,\dfrac{}{(\mathtt{STOP},\mathcal{G},9)\xrightarrow{(\tau,\emptyset)}(\not\Omega,\mathcal{G}[9\underset{11}{\mapsto}(\mathtt{MAIN},2.2)],11)}}{\mathit{State}\ 5\xrightarrow{(\tau,\emptyset)}\mathit{State}\ 6}$$

$$\text{where } \mathit{State}\ 6 = (\Omega\ \underset{\{\mathtt{b}\}}{\|}_{((\mathtt{MAIN},\Lambda),10,11)}\ \not\Omega,\mathcal{G}',1)$$

Figure 17: An example of computation with the tracking semantics in Fig. 15 and Fig. 16

6. CORRECTNESS

In this section we prove the correctness of the tracking semantics by showing that (i) the computations performed by the tracking semantics are equivalent to the computations performed by the standard semantics; and (ii) the graph produced by the tracking semantics is the track of the derivation. We also prove that the trace of a derivation can be automatically extracted from the track of this derivation. The proofs of these technical results can be found in Appendix A.

The first theorem shows that the computations performed with the tracking semantics are all and only the computations performed with the standard semantics. The only difference between them from an operational point of view is that the tracking semantics needs to perform one step when a `STOP` is evaluated (to add its specification position to the track) and then stops, while the standard semantics stops without performing any additional step.

We need first to define the concept of derivation strategy that is used to assure that the derivations with both semantics make the same decisions during a computation.

Definition 6 (Derivation strategy). *A derivation strategy Ψ is a deterministic function that given a process to be evaluated (a* control*), and given an operational semantics $\mathcal{O}$, it returns one rule in $\mathcal{O}$ to apply. We denote with $\mathcal{D}_\Psi$ a derivation that uses the derivation strategy Ψ to deterministically select an appropriate rule in each step of the derivation.*

Note that both the standard semantics (see Figure 9) and the tracking semantics (see Figure 15 and Figure 16) use the same control in all rules (except for `STOP`). Therefore, given a process P both semantics using the same derivation strategy will produce derivations with exactly the same number of rewriting steps (except for `STOP`), and the same sequences of controls.

We define now the conservativeness theorem.

Theorem 1 (Conservativeness). *Let $\mathcal{S}$ be a CSP specification, P a process in $\mathcal{S}$, and $\mathcal{D}_\Psi$ and $\mathcal{D}'_\Psi$ the derivations of P performed using the same derivation strategy Ψ with the standard semantics of CSP and with the tracking semantics, respectively. Then, the sequence of rules applied in $\mathcal{D}_\Psi$ and $\mathcal{D}'_\Psi$ is exactly the same except that $\mathcal{D}'_\Psi$ performs one rewriting step more than $\mathcal{D}_\Psi$ for each (sub)computation that finishes with* `STOP`*.*

The second theorem states the correctness of the tracking semantics by ensuring that the graph produced is the track of the computation.

Theorem 2 (Semantics correctness). *Let $\mathcal{S}$ be a CSP specification, $\mathcal{D}$ a derivation of $\mathcal{S}$ performed with the tracking semantics, and $\mathcal{G}$ the graph produced by $\mathcal{D}$. Then, $\mathcal{G}$ is the track associated with $\mathcal{D}$.*

Our last result states that the trace of a derivation can be extracted from its associated track. To prove it, we define first an order on the event nodes of a track that corresponds to the order in which they were generated by the tracking semantics.

Definition 7. *(Event node order) Given a track $\mathcal{G} = (N, E_c, E_s)$ and nodes $m, n \in N$ such that $l(m), l(n) \in \Sigma$, m is* smaller *than n, represented by $m \ll n$ iff $m' < n'$ where $(m, m'), (n, n') \in E_c$.*

Intuitively, an event node m is smaller than an event node n if and only if the successor of m has a reference smaller than the reference of the successor of n. The following lemma is also necessary to prove that the order in which events occur in a derivation is directly related with the order of Definition 7. In the following we consider an augmented version of derivation $\mathcal{D}$ which includes the event fired by the application of the rule. So, we can represent derivation $\mathcal{D}$ as $P_1 \overset{\Theta_1}{\underset{e_1}{\rightsquigarrow}} \dots \overset{\Theta_j}{\underset{e_j}{\rightsquigarrow}} P_{j+1}$.

Lemma 1. *Given a derivation $\mathcal{D} = P_1 \overset{\Theta_1}{\underset{e_1}{\rightsquigarrow}} \dots \overset{\Theta_j}{\underset{e_j}{\rightsquigarrow}} P_{j+1}$ of the tracking semantics, and the track $\mathcal{G} = (N, E_c, E_s)$ produced by $\mathcal{D}$, then $\forall e_i \in \Sigma, 1 \leq i \leq j$,*

- *$\exists n \in N$ such that $l(n) = e_i$, and*
- *$\exists (n, n') \in E_c$ such that $n' = n + 1$.*

Therefore, Lemma 1 ensures that the order of Definition 7 corresponds to the order in which the semantics generates the nodes, because each event is added to the graph together with a new fresh reference for the prefixing operator. Since references are generated incrementally, the occurrence of an event e will generate a reference which is less than the reference generated with a posterior event e'. With this order, we can easily define a transformation to extract a trace from a track based on the following proposition:

Proposition 1. *Given a track $\mathcal{G} = (N, E_c, E_s)$, the trace induced by $\mathcal{G}$ is the sequence of events $T = e_1, \dots, e_m$ that labels the associated sequence of nodes $T' = n_1, \dots, n_m$ (i.e., $\forall e_i \in T, n_i \in T'$, $1 \leq i \leq m, l(n_i) = e_i$ and $e_i \in \Sigma$) where:*

1. *$\forall n_i \in T',\ 0 < i < m,\ n_i \ll n_{i+1}$.*
2. *$\forall n \in N$ such that $l(n) \in \Sigma$, if $(\nexists n' \in N \mid (n, n') \in E_s)$, then $n \in T'$.*
3. *$\forall n \in N$ such that $l(n) \in \Sigma$, if $(\forall n' \in N \mid (n, n') \in E_s \land\ n' \ll n)$, then $n \in T'$.*

Theorem 3 (Track correctness). *Let $\mathcal{S}$ be a CSP specification, $\mathcal{D}$ a derivation of $\mathcal{S}$ produced by the sequence of events (i.e., the trace) $T = e_1, \ldots, e_m$, and $\mathcal{G}$ the track associated with $\mathcal{D}$. Then, there exists a function f that extracts the trace T from the track $\mathcal{G}$, i.e., $f(\mathcal{G}) = T$.*

Proof. Proposition 1 allows to trivially define a function f such that $f(\mathcal{G}) = T$ being $\mathcal{G}$ the track of a derivation $\mathcal{D}$, and being T the trace of the same derivation. For a track $\mathcal{G} = (N, E_c, E_s)$ we have that

$$f((n : ns), E_c, E_s) = \begin{cases} f((ns), E_c, E_s) & \text{if } (\exists n' \in N | (n, n') \in E_s \wedge n \ll n') \\ (l(n) : f((ns), E_c, E_s)) & \text{otherwise} \end{cases}$$

where list $(n : ns)$ corresponds to the set $\{n \in N \mid l(n) \in \Sigma\}$ ordered with respect to order $\ll$ of Definition 7. ∎

7. IMPLEMENTATION

We have 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. CSP-Tracker incorporates mechanisms to produce coloured graphs that represent the tracks in a very intuitive way. CSP-Tracker implements the instrumented semantics in Figure 16, and thus it can generate the track of a (partial) derivation until it finishes or is stopped. In CSP-Tracker, both the tracking and slicing processes are completely automatic. Once the user has loaded a CSP specification, they can (automatically) produce a derivation and the tool internally generates the associated track. Then, the tool asks for the number of occurrence of the slicing criterion they are 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*[11] graphs. Figure 18 shows a screenshot of an interface of the tool showing the track and the trace of the specification in Example 1.

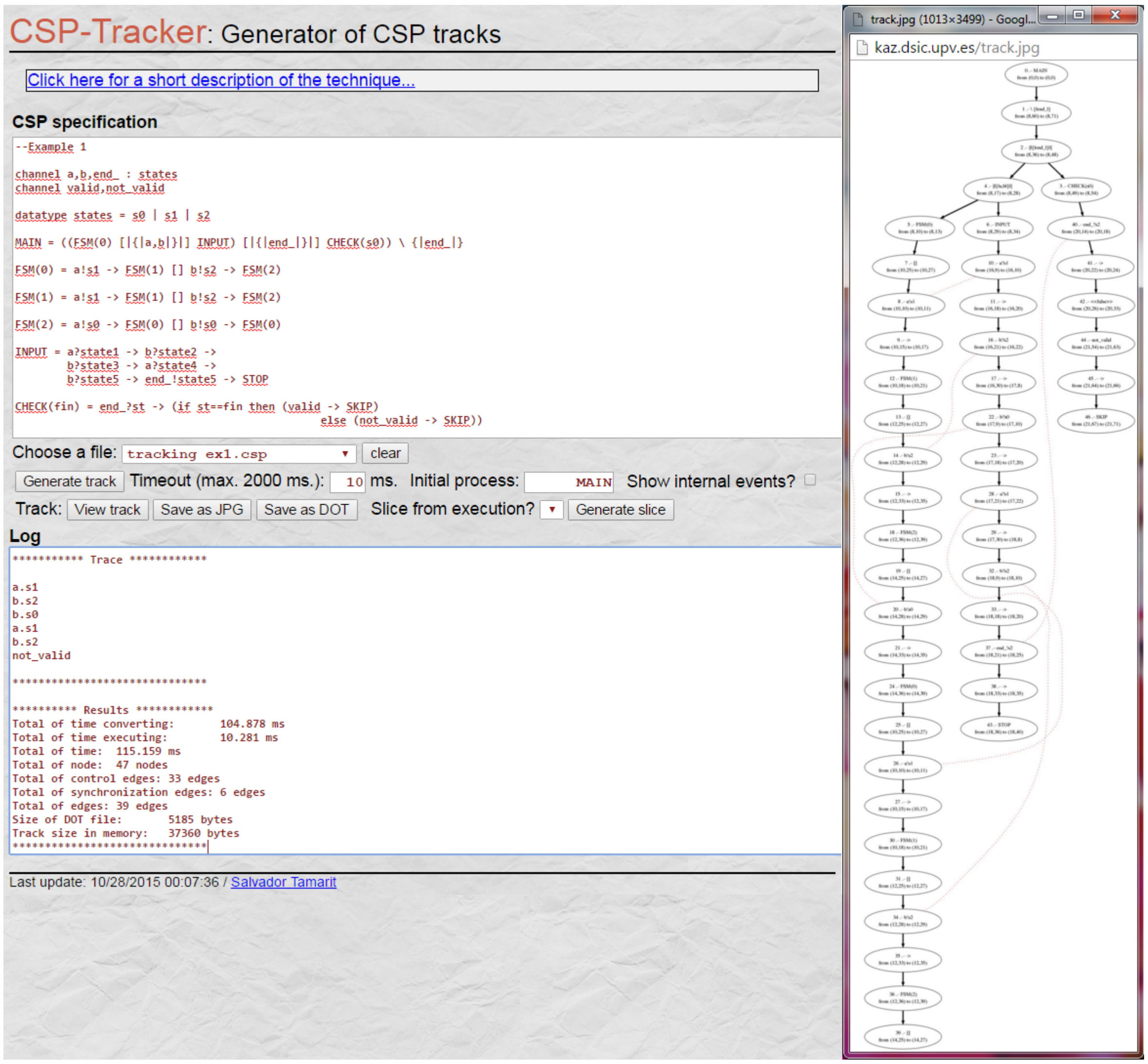
7.1. Architecture of CSP-Tracker

The information collected by CSP-Tracker is *dynamic*, and thus the subsequent analyses performed are very precise. We would like to allow the user to combine this information with other analyses that already exist for CSP. Therefore, we have integrated CSP-Tracker with another tool called SOC [32] able to perform different *static* analyses such as static slicing. Both tools are complementary (SOC generates static information while tracks provide dynamic information), and together can be useful, e.g., in debugging and program comprehension.

While SOC was implemented in Prolog, CSP-Tracker has been implemented in Erlang[12]. The election of Erlang was very conscious because Erlang is one of the most efficient languages for the use of multiple threads and parallelism [57, 14]; 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.

All modules except the parser and the graph generator were implemented in Erlang. The CSP parser translates CSP to a Prolog representation that can be used by SOC. This parser is part of ProB [30, 31] which is one of the most extended IDE for CSP. Once CSP is translated to Prolog, we use a Prolog module to translate the resulting code to an Erlang structure. This last 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.

[11]`http://www.graphviz.org/`
[12]`http://www.erlang.org/`


CSP-Tracker: Generator of CSP tracks

Click here for a short description of the technique...

CSP specification

```
--Example 1

channel a,b,end_ : states
channel valid,not_valid

datatype states = s0 | s1 | s2

MAIN = ((FSM(0) [|{|a,b|}|] INPUT) [|{|end_|}|] CHECK(s0)) \ {|end_|}

FSM(0) = a!s1 -> FSM(1) [] b!s2 -> FSM(2)

FSM(1) = a!s1 -> FSM(1) [] b!s2 -> FSM(2)

FSM(2) = a!s0 -> FSM(0) [] b!s0 -> FSM(0)

INPUT = a?state1 -> b?state2 ->
        b?state3 -> a?state4 ->
        b?state5 -> end_!state5 -> STOP

CHECK(fin) = end_?st -> (if st==fin then (valid -> SKIP)
                                    else (not_valid -> SKIP))
```

Choose a file: tracking ex1.csp

Timeout (max. 2000 ms.): 10 ms. Initial process: MAIN Show internal events? ☐

Track: Slice from execution?

Log

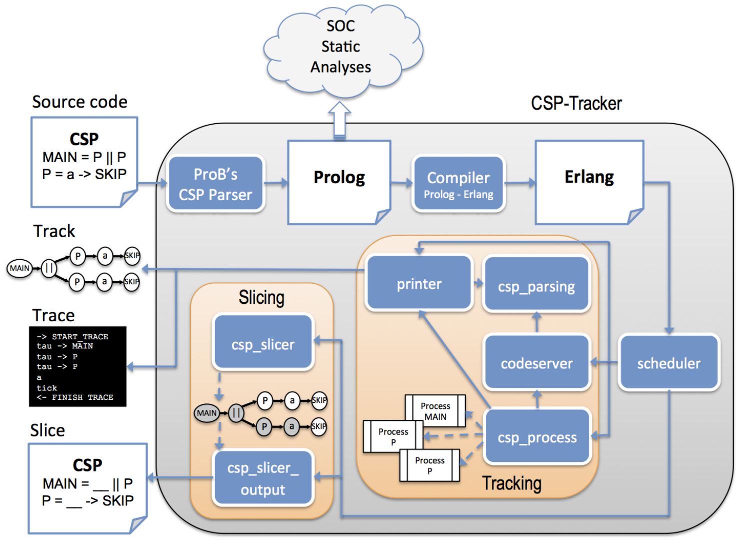
```
*********** Trace ************

a.s1
b.s2
b.s0
a.s1
b.s2
not_valid

******************************

********** Results ************
Total of time converting:       104.878 ms
Total of time executing:        10.281 ms
Total of time:  115.159 ms
Total of node:  47 nodes
Total of control edges: 33 edges
Total of synchronization edges: 6 edges
Total of edges: 39 edges
Size of DOT file:       5185 bytes
Track size in memory:   37360 bytes
******************************
```

Last update: 10/28/2015 00:07:36 / Salvador Tamarit

Figure 18: Track of a CSP specification produced by CSP-Tracker

Figure 19 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`: It translates a CSP specification into a Prolog representation. This Prolog structure acts as an intermediate language that is used by SOC to perform complementary 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 20. For instance, node 5 with literal `b` has the specification position `from (4,10) to (4,11)`; which means that `b` appears in the source code between columns 10 and 11 of line 4.
- `Compiler Prolog-Erlang`: It produces an Erlang representation equivalent to the Prolog structure.
- `scheduler`: This module initializes 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 been generated, it traverses the track from the slicing criterion to extract the slice.



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

Figure 19: CSP-Tracker's Architecture

- `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 then calls `codeserver` to create a new process that represents `P`.
- `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. For instance, see the specification in `https://github.com/mistupv/csp_tracker/blob/dac20d6da0235cc0d217ecc4f6bc6eb803243651/benchmarks/ABPSlice.csp#L80`, where line 80 specifies a slicing criterion with the fresh channel `slice`, which is declared in line 35. 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.
- `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 specifica-

tion by STOP. Hence, two outputs are possible: (i) a non-executable part of the specification useful for debugging, or also (ii) a well-formed CSP specification able to produce the same computation until the slicing criterion is reached.

7.2. Using CSP-Tracker

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.html

This section shows the use of CSP-Tracker's web interface with three illustrative scenarios. The first scenario shows a specification that successfully finishes, the second scenario shows the case where the program is deadlocked and the third shows the case where the program produces an infinite computation.

In the first scenario, we consider a simple modification of Example 3, where all STOP terms have been replaced by SKIP. This change makes the process finish successfully. We just load the first example and press the button Generate Track:

```
Creating the Erlang representation of the CSP file...
...
Created.

-> START_TRACE

   tau -> Call to process MAIN
   tau -> Call to process P
a
   tau
   tau
   tick

<- FINISH_TRACE
```

Once the execution is finished, a file track.pdf is produced containing the track associated to the execution of the CSP specification. The track generated by CSP-Tracker for this example is shown in Figure 20.

During the execution, the trace produced is shown in the Log window. In the previous case, this trace was formed by both the internal and the external events fired by the semantics. This is interesting for a programmer that wants to analyse the behavior of CSP from a semantic point of view. However, the conventional programmer is only interested in the usual trace, only formed by external events. This can be obtained by unchecking the option Show internal events?:

```
-> START_TRACE

a

<- FINISH_TRACE
```

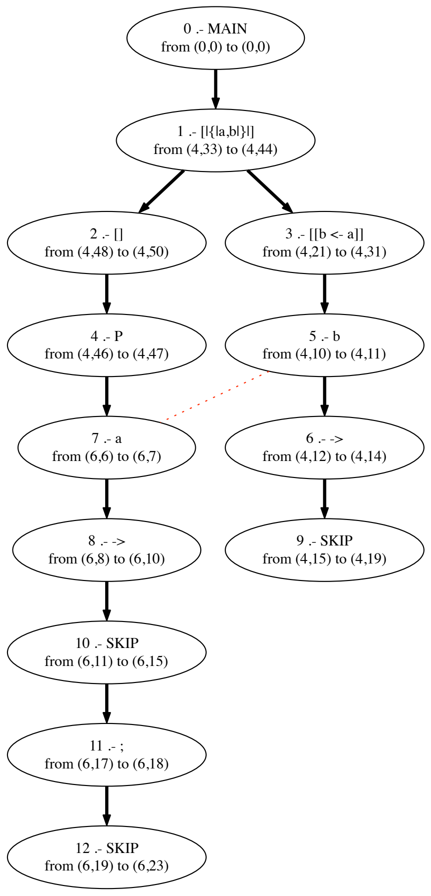


Figure 20: A track generated by CSP-Tracker

Our second scenario is Example 1, that produces a deadlock. In this case, the tool automatically detects the deadlock and produces the trace until the deadlock happens. We have the following trace:

```
-> START_TRACE

a.s1
b.s2
b.s0
a.s1
b.s2
notvalid

<- STOPPED_TRACE (deadlock)
```

Our third scenario is Example 6 that produces an infinite loop. However, this is not a problem for the tracker, that can still run the specification and generate the track until it is stopped or a timeout is reached. We have the following trace:

```
-> START_TRACE

a
b
...
a
b

Timeout.
```

7.2.1. Selecting computations

CSP-Tracker uses the standard CSP semantics to produce computations. This means that computations are non-deterministic by construction. However, the user may be interested in some particular execution that she/he wants to reproduce (e.g., because it is buggy, or just to produce a slice and study one particular interleaving). There exist techniques to select a concrete computation using techniques such as forcing a path (see, e.g., [52]). One of them is to create a parallel synchronization of two processes: (i) a deterministic process that can only execute the desired trace (for instance, `TRACE = a -> b -> SKIP`), and (ii) the process that we want to behave as that trace.

Example 8. *Consider again the non-deterministic CSP specification of the FSM in Figure 1:*

```
channel a, b, s0, s1, s2
FSM0 = s0 -> (a -> FSM1 [] b -> FSM2)
FSM1 = s1 -> (a -> FSM1 [] b -> FSM2)
FSM2 = s2 -> (a -> FSM0 [] b -> FSM0)
```

By just adding the following two lines:

$$\texttt{MAIN = (FSM0} \underset{\{\texttt{s0,s1,s2,a,b}\}}{||} \texttt{TRACE)}$$

```
TRACE = s0 -> b -> s2 -> a -> s0 -> SKIP
```

the specification becomes deterministic and it always produces the same trace: $\langle \texttt{s0}, \texttt{b}, \texttt{s2}, \texttt{a}, \texttt{s0} \rangle$.

8. EMPIRICAL EVALUATION

Once we have theoretically proved the correctness of the technique, in this section we conduct a series of experiments in order to evaluate the scalability of our implementation (CSP-Tracker). CSP-Tracker is an augmented CSP interpreter instrumented for tracking and, thus, it introduces an overhead in the computations when generating tracks. In the following, we evaluate the time needed to generate a collection of tracks, and the size of these tracks, to provide a precise and quantitative idea of the performance of the execution and track generation in practice.

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 benchmarks have been collected from different repositories and articles ensuring that they (together) cover all the syntax, and also that they have a wide range of possible executions (finite executions and infinite executions with both finite and infinite nested parallelism). The source code of the benchmarks can be found at: `https://github.com/mistupv/csp_tracker/tree/master/benchmarks`. In each benchmark, we have included a header describing it and providing details about the authors and a reference to its source.

In order to evaluate the performance of our tool, we strictly followed the methodology proposed in [17, 51]. 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 producing different derivations due to non-determinism.

As we can have infinite executions (e.g. livelock processes) we need a way to automatically stop them. For this reason, we use a timeout of 2 seconds. This threshold 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 also 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, and synchronizations, etc.) executed exactly the same time.

Table 1: Benchmark results showing CSP-Tracker performance

Benchmark	CSP2Erlang (ms)	Generate Track (ms)	Total (ms)
ABP.csp	[$_{201.87}$ 202.91 $_{203.94}$]	[$_{2005.99}$ 2006.34 $_{2006.70}$]	[$_{2208.16}$ 2209.25 $_{2210.34}$]
ATM.csp	[$_{314.66}$ 315.97 $_{317.28}$]	[$_{314.19}$ 374.21 $_{434.23}$]	[$_{630.17}$ 690.18 $_{750.19}$]
Buses.csp	[$_{125.52}$ 126.30 $_{127.09}$]	[$_{0.88}$ 0.89 $_{0.89}$]	[$_{126.40}$ 127.19 $_{127.97}$]
CPU.csp	[$_{181.06}$ 181.82 $_{182.59}$]	[$_{8.85}$ 8.92 $_{8.98}$]	[$_{189.97}$ 190.74 $_{191.51}$]
Disk.csp	[$_{185.86}$ 186.89 $_{187.92}$]	[$_{23.12}$ 23.21 $_{23.29}$]	[$_{209.07}$ 210.10 $_{211.13}$]
Loop.csp	[$_{126.19}$ 127.09 $_{127.98}$]	[$_{2006.55}$ 2006.90 $_{2007.26}$]	[$_{2133.02}$ 2133.99 $_{2134.96}$]
Oven.csp	[$_{203.84}$ 204.82 $_{205.79}$]	[$_{33.98}$ 37.10 $_{40.22}$]	[$_{238.64}$ 241.92 $_{245.20}$]
ProdCons.csp	[$_{127.77}$ 128.55 $_{129.33}$]	[$_{2006.54}$ 2006.88 $_{2007.22}$]	[$_{2134.59}$ 2135.43 $_{2136.27}$]
ReadWrite.csp	[$_{143.40}$ 144.26 $_{145.12}$]	[$_{2005.10}$ 2005.45 $_{2005.80}$]	[$_{2148.76}$ 2149.71 $_{2150.65}$]
Traffic.csp	[$_{158.56}$ 159.43 $_{160.29}$]	[$_{6.43}$ 6.93 $_{7.42}$]	[$_{165.34}$ 166.35 $_{167.36}$]
Average	[$_{176.87}$ 177.80 $_{178.73}$]	[$_{841.16}$ 847.68 $_{854.20}$]	[$_{1018.41}$ 1025.49 $_{1019.76}$]

(a) Execution time results

Benchmark	#Nodes	#Control Edges	#Sync. Edges
ABP.csp	[$_{1505.61}$ 1506.17 $_{1506.73}$]	[$_{1130.49}$ 1130.91 $_{1131.33}$]	[$_{172.61}$ 172.72 $_{172.84}$]
ATM.csp	[$_{364.09}$ 405.64 $_{447.19}$]	[$_{242.64}$ 269.10 $_{295.57}$]	[$_{58.10}$ 65.57 $_{73.04}$]
Buses.csp	[$_{22.00}$ 22.00 $_{22.00}$]	[$_{15.00}$ 15.00 $_{15.00}$]	[$_{3.00}$ 3.00 $_{3.00}$]
CPU.csp	[$_{87.43}$ 87.76 $_{88.09}$]	[$_{62.02}$ 62.23 $_{62.44}$]	[$_{9.21}$ 9.27 $_{9.33}$]
Disk.csp	[$_{148.50}$ 148.74 $_{148.98}$]	[$_{101.68}$ 101.83 $_{101.98}$]	[$_{21.91}$ 21.95 $_{22.00}$]
Loop.csp	[$_{1537.53}$ 1538.34 $_{1539.14}$]	[$_{924.11}$ 924.59 $_{925.08}$]	[$_{305.94}$ 306.10 $_{306.27}$]
Oven.csp	[$_{157.16}$ 163.37 $_{169.59}$]	[$_{109.56}$ 113.59 $_{117.63}$]	[$_{53.12}$ 55.74 $_{58.35}$]
ProdCons.csp	[$_{1535.43}$ 1536.09 $_{1536.75}$]	[$_{922.38}$ 922.77 $_{923.17}$]	[$_{305.70}$ 305.84 $_{305.98}$]
ReadWrite.csp	[$_{1475.85}$ 1476.56 $_{1477.28}$]	[$_{1030.85}$ 1031.45 $_{1032.06}$]	[$_{221.62}$ 221.89 $_{222.16}$]
Traffic.csp	[$_{61.18}$ 64.37 $_{67.56}$]	[$_{43.01}$ 45.01 $_{47.01}$]	[$_{4.72}$ 5.12 $_{5.52}$]
Average	[$_{689.478}$ 694.90 $_{700.33}$]	[$_{458.17}$ 461.65 $_{465.13}$]	[$_{115.59}$ 116.72 $_{117.85}$]

(b) Track size results

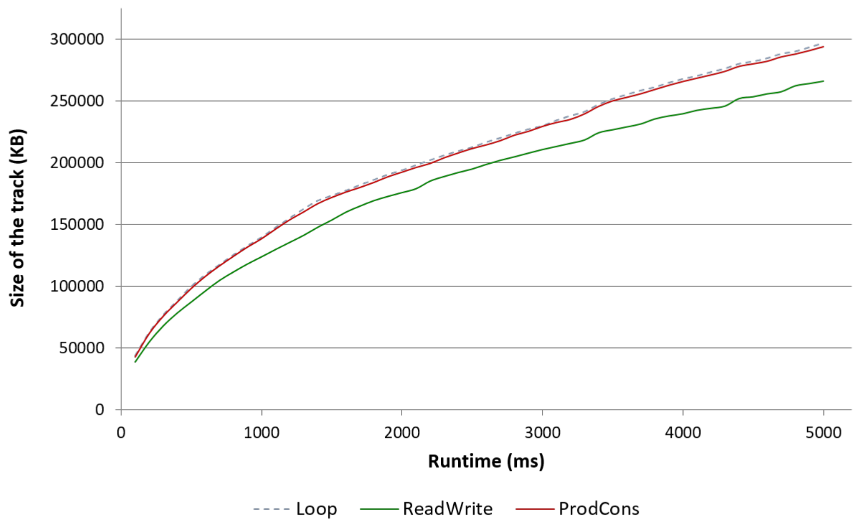
Benchmark	Memory Size (Kbytes)
ABP.csp	[$_{172.98}$ 173.05 $_{173.11}$]
ATM.csp	[$_{42.61}$ 47.56 $_{52.51}$]
Buses.csp	[$_{2.43}$ 2.43 $_{2.43}$]
CPU.csp	[$_{9.59}$ 9.63 $_{9.67}$]
Disk.csp	[$_{16.72}$ 16.74 $_{16.77}$]
Loop.csp	[$_{191.42}$ 191.53 $_{191.63}$]
Oven.csp	[$_{20.03}$ 20.86 $_{21.69}$]
ProdCons.csp	[$_{189.44}$ 189.53 $_{189.61}$]
ReadWrite.csp	[$_{171.57}$ 171.66 $_{171.76}$]
Traffic.csp	[$_{6.44}$ 6.79 $_{70.30}$]
Average	[$_{82.32}$ 82.98 $_{83.63}$]

(c) Memory size results

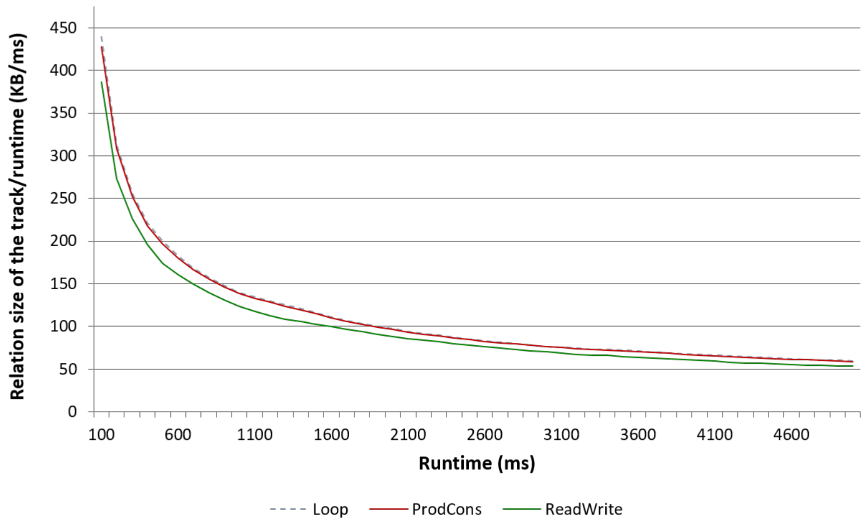
To ensure real independence, the first iteration was always discarded (to avoid influence of dynamically loaded libraries persisting in physical memory, data persisting in the disk cache, etc.). Thus, we obtained 1000 statistical values. Then, we computed the 0.99 confidence interval across the computed values from the different 1000 executions.

This process was repeated for the 10 benchmarks, 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.

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. Each column in the tables has the following meaning: `Benchmark` is the name of the benchmark. `CSP2Erlang` is the time needed to compile the CSP program to an equivalent Erlang program, `Generate track` is the time needed to generate a track from the execution of the Erlang program (with a timeout of 2 seconds), and `Total` is the sum of the previous two measures. It represents



(a) Track size with respect to runtime



(b) Track growth rate with respect to runtime

Figure 21: Trend rate of the track growth

the total amount of time needed by CSP-Tracker to produce the track from the execution of a CSP benchmark. `#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. `Memory Size` is the size in the hard disk of the track generated (stored as a .dot file). Times are measured in milliseconds and memory sizes are represented in Kilobytes.

In the tables we can see that generating a track 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 is one order of magnitud less than track generation in those cases where the threshold is reached. Regarding column `Memory Size` we can see that it is as an average lower than 85 KB, memory consumption is small even for big tracks (big number of nodes, and control and synchronization edges).

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 21(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 21(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.

9. CONCLUSIONS

This work formalizes the notion of track. A track can be seen as a of an execution enhanced with information about source positions. Therefore, it allows for identifying what parts of the source code have been executed and in what order.

The main results of this work are:

1. A formalization of tracks. Tracks are defined as graphs that explicitly show what events happen sequentially, what events happen in parallel and what events are synchronized. They are computed dynamically to represent one specific execution.

2. A definition of the first tracking semantics for CSP. We have defined a formal operational semantics instrumented for tracking. The execution of the tracking semantics produces (as a side effect) the track of the computation. This track is generated step by step by the semantics, and thus, it can be also used to produce a track of an infinite computation until it is stopped.

3. A proof that the trace of a computation can be extracted from the track of this computation (Theorem 3). This result shows that a track is more informative than a trace because the former contains the later but, also, tracks include explicit information that relate the trace with the source code. Therefore, our instrumented semantics could serve as a theoretical foundation to relate the semantics of a specification with its track, and to define and prove properties of analyses based on both, traces and tracks.

4. An implementation of the first CSP tracker and its empirical evaluation. We have implemented a tool called CSP-Tracker able to automatically generate the tracks of a CSP specification. It also includes the first dynamic slicer for CSP. It is based on tracks, and it allows the programmer to select a (set of) point in the specification or in the track, and automatically extract the part of the specification needed to reach this point.

Tracks can be useful not only for tracking computations but for debugging and program comprehension. We have illustrated both applications with use cases. On the practical side, we have empirically evaluated our implementation. The main conclusion is that tracking is scalable to big systems because generating a track is a very efficient task, and the memory consumption is acceptable (it follows a function $y = x^{1/2}$, where y represents the size of the track and x represents the runtime). If the number of processes is huge, the track can be stored in secondary memory (e.g., with every GB of nodes generated). Moreover, 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.).

As a future work, we plan to improve our tool. Currently, it is able to produce (backward) slices from both the track of a computation and its CSP specification. We want to enhance this behaviour with a track viewer tool to highlight the parts of the code that are executed at each step. Moreover, we plan to augment the slicing functionality, producing:

- Forward slices, which allows the user to specify a slicing criterion (e.g., an event) and extract the part of the track/specification that was executed after this event; or also, that was affected by this event (this event was needed to execute the slice).
- Chops, which are slices produced from two points in the specification. They contain those parts of the specification that are executed after the first point and before the second point.

References

[1] Abdallah, A.E., Jones, C.B., Sanders, J.W. (eds.): Communicating Sequential Processes. LNCS, vol. 3525. Springer, Heidelberg (2005)

[2] Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press (1998)

[3] Brassel, B., Hanus, M., Huch, F., Vidal, G.: A Semantics for Tracing Declarative Multi-paradigm Programs. In: Moggi, E., Warren, D.S. (eds.), 6th ACM SIGPLAN Int'l Conf. on Principles and Practice of Declarative Programming (PPDP'04), pp. 179–190. ACM, New York, NY, USA (2004)

[4] Broadfoot, G.H., Hopcroft, P.J.: A Paradigm Shift in Software Development. In: Proceedings of Embedded World Conference, Nuremberg (2012)

[5] Brown, N.C.C. and Smith, M.L.: Representation and Implementation of CSP and VCR Traces. In: Welch, P.H., Stepney et al. (eds.) The thirty-first Communicating Process Architectures Conference (CPA'08), Concurrent Systems Engineering Series, vol. 66, pp. 329–345. (2008)

[6] 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)

[7] Brückner, I., Wehrheim, H.: Slicing CSP-OZ Specifications for Verification. Technical report, SFB/TR 14 AVACS (2005) Accessible via `http://www.avacs.org/`

[8] Callahan, D. and Sublok, J.: Static Analysis of Low-level Synchronization. In: 1988 ACM SIGPLAN and SIGOPS workshop on Parallel and Distributed Debugging (PADD'88), pp. 100–111. ACM (1988)

[9] Carter, J.D., Gardner, W.B.: A Formal CSP Framework for Message-Passing HPC Programming. In: IEEE Canada 19th Annual Canadian Conference on Electrical and Computer Engineering (CCECE 2006), pp. 944-948. (2006)

[10] Carvalho, G., Falcão, D., Mota, A., Sampaio, A.: A Process Algebra Based Strategy for Generating Test Vectors from SCR Specifications. In: 15th Brazilian Symposium on Formal Methods: Foundations and Applications (SBMF'12). LNCS, vol. 7498, pp. 67–82. Springer, Heidelberg (2012)

[11] 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 CSL (2001)

[12] Chitil, O., Runciman, C., Wallace, M.: Transforming Haskell for Tracing. In: Peña, R., Arts, T. (eds.) IFL 2002, Revised Selected Papers. LNCS, vol. 2670, pp. 165–181. Springer, Heidelberg (2003)

[13] Chitil, O., Lou, Y.: Structure and Properties of Traces for Functional Programs. Electronic Notes in Theoretical Computer Science (ENTCS). 176(1), 39–63 (2007)

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

[15] 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)

[16] Gibson-Robinson, T., Armstrong, P., Boulgakov, A., Roscoe, A.W.: FDR3 – A Modern Model Checker for CSP. In: Tools and Algorithms for the Construction and Analysis of Systems. LNCS, vol.8413, pp. 187–201. Springer, Heidelberg (2014)

[17] Georges, A., Buytaert, D., Eeckhout, L.: Statistically Rigorous Java Performance Evaluation. In: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications (OOPSLA'07), pp. 57–76. ACM, NY, USA (2009)

[18] Goswami, D. and Mall, R.: Fast Slicing of Concurrent Programs. In: Banerjee, P., Prasanna, V.K. and Sinha, B.P. (eds.) 6th Int'l Conf. on High Performance Computing (HiPC'99). LNCS, vol. 1745, pp. 38–42. Springer, Heildeberg (1999)

[19] Gruian, F., Schoeberl, M.: Hardware support for CSP on a Java chip multiprocessor. Microprocessors and Microsystems - Embedded Hardware Design. 37(4-5), 472–481 (2013)

[20] Gunter, E.L., Yasmeen, A., Gunter, C.A., Nguyen, A.: Specifying and Analyzing Workflows for Automated Identification and Data Capture. In: 42nd Hawaii Int'l Conf. on System Sciences (HICSS'09), pp. 1–11. IEEE Computer Society (2009)

[21] Hall, R.J.: Automatic Extraction of Executable Program Subsets by Simultaneous Dynamic Program Slicing. Automated Software Engineering. 2(1), 33–53. Kluwer Academic Publishers (1995)

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

[23] Jeong, H.Y., Hong, B.H.: CSP Based E-Learning Model in Cloud Computing Environment. In: T.-h. Kim et al. (eds.) Computer Applications for Graphics, Grid Computing, and Industrial Environment (GDC/IESH/CGAG'12), CCIS, vol. 351, pp. 110–117. Springer, Heildeberg (2012)

[24] Jeong, H.Y., Hong, B.H.: CSP Based Relation Structure for Social Network Service. In: T.-h. Kim et al. (eds.) Computer Applications for Software Engineering, Disaster Recovery, and Business Continuity (ASEA/DRBC'12), CCIS, vol. 340, pp. 163–170. Springer, Heildeberg (2012)

[25] 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'95), vol. 2 (Software Technology), pp. 516–524. IEEE Computer Society, Washington, DC, USA (1995)

[26] Krinke, J.: Context-Sensitive Slicing of Concurrent Programs. ACM SIGSOFT Software Engineering Notes. 28(5) (2003)

[27] Krinke, J.: Barrier Slicing and Chopping. In: 3rd IEEE Int'l Workshop on Source Code Analysis and Manipulation (SCAM'03), pp. 81–87. IEEE Computer Society, Washington, DC, USA (2003)

[28] Ladkin, P., Simons, B.: Static Deadlock Analysis for CSP-Type Communications. Responsive Computer Systems (Chapter 5), Kluwer Academic Publishers (1995)

[29] Lecomte, T., Burdy, L., Leuschel, M.: Formally Checking Large Data Sets in the Railways. CoRR abs/1210.6815 (2012)

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

[31] 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)

[32] 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)

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

[34] Li, J., He, J., Zhu, H., Pu, G.: Modeling and Verifying Web Services Choreography Using Process Algebra. In: 31st IEEE Software Engineering Workshop (SEW'07), pp. 256–268. IEEE Computer Society (2007)

[35] Li, L., Gunter, E.L., Mansky, W.: Symbolic Semantics for CSP. Computer Science Research and Tech Reports (2013)

[36] Llorens, M., Oliver, J., Silva, J., Tamarit, S.: A Tracking Semantics for CSP. In: 10th Int'l Conf. on Mathematics of Program Construction (MPC 2010). LNCS, vol. 6120, pp. 248–270. Springer, Heildeberg (2010)

[37] Lowe, G.: Breaking and Fixing the Needham-Schroeder Public-Key Protocol Using FDR. In: Margaria, T., Steffen, B. (eds.), Second Int'l Workshop Tools and Algorithms for Construction and Analysis of Systems (TACAS'96). LNCS, vol. 1055, pp. 147–166. Springer, Heidelberg (1996)

[38] McInnes, A.I: Modeling and Analysis of TinyOS Sensor Node Firmware: a CSP Approach. ACM Trans. on Embedded Computing Systems, vol. 12(1), pp. 5:1–5:23. ACM, New York, USA (2011).

[39] Milner, R.: A Calculus of Communicating Systems. LNCS, vol. 92. Springer, Berlin (1980)

[40] Milner, R.: Communication and Concurrency. Prentice Hall (1989)

[41] Moran, M., Heather, J., Schneider, S.: Verifying Anonymity in Voting Systems Using CSP. Formal Aspects of Computing, pp. 1–36. Springer-Verlag (2012)

[42] O'Halloran, C.: Session II-A: Verification and Validation/High-Assurance Systems Acceptance-Based Assurance. In: 26th IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE'01), pp. 63. IEEE Computer Society (2001)

[43] Palikareva, H., Ouaknine. J., Roscoe, A.W.: SAT-solving in CSP trace refinement. Science of Computer Programming. 77, 10–11 (2012)

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

[45] Roscoe, A.W.: The pursuit of buffer tolerance. Unpublished manuscript (2005) Obtainable from `web.comlab.ox.ac.uk/oucl/work/bill.roscoe/publications/106.pdf`

[46] Roscoe, A.W.: Confluence thanks to extensional determinism. In: Proceedings of Bertinoro meeting on Concurrency, BRICS 2005. Revised version, publication reference ENTCS 1336 (2006)

[47] Roscoe, A.W.: Understanding Concurrent Systems. Springer-Verlag, London (2010)

[48] Saifhashemi, A., Beerel, P. A.: High Level Modeling of Channel-Based Asynchronous Circuits Using Verilog. In: The 28th Communicating Process Architectures Conf. (CPA'05), pp. 275–288. IOS Press (2005)

[49] Silva, J.: A vocabulary of program slicing-based techniques. ACM Computing Surveys. 44(3)-num.12 (2012)

[50] Smith, M.L. Focusing on traces to link VCR and CSP. In: East, J.M., Welch, P., Duce, D., and Green, M. (eds.), The 27th Communicating Process Architectures Conference (CPA'04), pp. 353–360. IOS Press (2004)

[51] Souilah, I., Francalanza, A., Sassone, V.: A Formal Model of Provenance in Distributed Systems. In: First Works. on Theory and Practice of Provenance (TAPP'09), pp. 1–11. USENIX Assoc. (2009)

[52] Taylor R.N., Levine D.L., Kelly C.D.: Structural testing of concurrent programs. IEEE Transactions on Software Engineering. 18(3), 206–215 (1992)

[53] The Go Project. Language Design FAQ: Why build concurrency on the ideas of CSP? `https://golang.org/doc/faq#csp`. Retrieved on March 2016.

[54] Tip, F.: A survey of program slicing techniques. Journal of Programming Languages. 3, 121–189 (1995)

[55] Wang, X.: Maximal Confluent Processes. In: Application and Theory of Petri Nets. LNCS vol. 7347, pp. 188–207. Springer, Heidelberg (2012)

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

[57] Zhao, X., Jamali, N.: Supporting Deadline Constrained Distributed Computations on Grids. In: 12th IEEE/ACM Int'l Conf. on Grid Computing (GRID'11), pp. 165–172. IEEE Computer Society (2011)

Appendix A. Proofs of Technical Results. CORRECTNESS.

In this section we prove the correctness of the tracking semantics by showing that (i) the computations performed by the tracking semantics are equivalent to the computations performed by the standard semantics; and (ii) the graph produced by the tracking semantics is the track of the derivation. We also prove that the trace of a derivation can be automatically extracted from the track of this derivation.

The first theorem shows that the tracking semantics is a conservative extension with respect to the standard semantics, in the sense that the computations performed with the tracking semantics are all and only the computations performed with the standard semantics. The only difference between them from an operational point of view is that the tracking semantics needs to perform one step when a `STOP` is evaluated (to add its specification position to the track) and then stops, while the standard semantics stops without performing any additional step.

We need first to define the concept of derivation strategy that is used to assure that the derivations with both semantics make the same decisions during a computation.

Definition 6. *(Derivation strategy) A derivation strategy Ψ is a deterministic function that given a process to be evaluated (a* control*), and given an operational semantics $\mathcal{O}$, it returns one rule in $\mathcal{O}$ to apply. We denote with $\mathcal{D}_\Psi$ a derivation that uses the derivation strategy Ψ to deterministically select an appropriate rule in each step of the derivation.*

Note that both the standard semantics (see Figure 9) and the tracking semantics (see Figure 15 and Figure 16) use the same control in all rules (except for `STOP`). Therefore, given a process P both semantics using the same derivation strategy will produce derivations with exactly the same number of rewriting steps (except for `STOP`), and the same sequences of controls.

We define now the conservativeness theorem.

Theorem 1 (Conservativeness). *Let $\mathcal{S}$ be a CSP specification, P a process in $\mathcal{S}$, and $\mathcal{D}_\Psi$ and $\mathcal{D}'_\Psi$ the derivations of P performed using the same derivation strategy Ψ with the standard semantics of CSP and with the tracking semantics, respectively. Then, the sequence of rules applied in $\mathcal{D}_\Psi$ and $\mathcal{D}'_\Psi$ is exactly the same except that $\mathcal{D}'_\Psi$ performs one rewriting step more than $\mathcal{D}_\Psi$ for each (sub)computation that finishes with* `STOP`.

Proof. Firstly, rule (`STOP`) of the tracking semantics is the only rule that is not present in the standard semantics. When a `STOP` is reached in a derivation, the standard semantics stops the (sub)computation because no rule is applicable. In the tracking semantics, when a `STOP` is reached in a derivation, the only rule applicable is (`STOP`) which performs τ and puts Ω in the control:

$$\frac{}{(\mathtt{STOP}_\alpha, \mathcal{G}, m) \xrightarrow{(\tau,\emptyset)} (\Omega, \mathcal{G}[m \underset{n}{\mapsto} \alpha], n)}$$

Then, the (sub)computation is stopped because no rule is applicable for Ω. Therefore, when the control in the derivation is `STOP`, the tracking semantics performs one additional rewriting step with rule (`STOP`).

The claim follows from the fact that both semantics have exactly the same number of rules except for rule (`STOP`), and these rules have the same control in all the states of the rules (thus the tracking semantics is a conservative extension of the standard semantics). Therefore, all derivations in both semantics have exactly the same number of steps and they are composed of the same sequences of rewriting steps except for (sub)derivations finishing with `STOP` where the tracking semantics performs one rewriting step more (applying rule (`STOP`)). ■

The second theorem states the correctness of the tracking semantics by ensuring that the graph produced is the track of the computation. To prove this theorem, the following lemmas are used.

Lemma 2. *Let $\mathcal{S}$ be a CSP specification, $\mathcal{D}$ a complete derivation of $\mathcal{S}$ performed with the tracking semantics, and $\mathcal{G}$ the graph produced by $\mathcal{D}$. Then, for each prefixing ($co \to P$) in the control of the left state of a rewriting step in $\mathcal{D}$, we have that $\mathcal{P}os(co)$ and $\mathcal{P}os(\to)$ are nodes of $\mathcal{G}$ and $\mathcal{P}os(\to)$ is the successor of $\mathcal{P}os(co)$.*

Proof. If a prefixing $co \rightarrow P$ is in the control of the left state of a rewriting step, following the tracking semantics, only the rule (Prefixing) can be applied. By definition of this rule (see Figure 16), m is the reference of the current node in $\mathcal{G}$. Rule (Prefixing) adds two new nodes to the graph: n and p. The node m is labeled with the specification position of complex object co and has successor n. The node n is labeled with the specification position of operator $\rightarrow$ and has parent m and successor p (a fresh reference). Therefore, we have that $\mathcal{P}os(co)$ and $\mathcal{P}os(\rightarrow)$ are nodes of $\mathcal{G}$ and $\mathcal{P}os(\rightarrow)$ is the successor of $\mathcal{P}os(co)$. ■

Lemma 3. *Let $\mathcal{S}$ be a CSP specification and $\mathcal{D}$ a derivation of $\mathcal{S}$ performed with the tracking semantics. Then, it holds that $last'(R) = last'(R')$ for each rewriting step $R \overset{\Theta}{\rightsquigarrow} R'$ in $\mathcal{D}$ with $R' \neq \Omega$ and $R' \neq \not\Omega$.*

Proof. We prove this lemma by induction on the length of Θ. In the base case, Θ is empty, and thus only the rules (Process Call), (Prefixing), (Internal Choice 1 and 2), (Conditional Choice 1 and 2) and (Synchronized Parallelism 4) can be applied. In all cases, the lemma holds trivially by the definition of $last'$. We assume as the induction hypothesis that the lemma holds for a non-empty Θ_i with $i > 0$ rewriting steps; and prove that the lemma also holds for a Θ_{i+1} with $i + 1$ rewriting steps. We can assume that $\Theta_{i+1} = R \overset{\Theta'}{\rightsquigarrow} R'$, thus, we have to prove that the lemma holds for any possible R and R'. The possible cases are the following:

(External Choice 1 and 2) This case is trivial because the specification positions of R and R' are the same. Hence, $last'(R) = last'(R')$.

(External Choice 3 and 4) Both cases are similar. Thus, we only discuss (External Choice 3). In the case of (External Choice 3), $last'(P_1 \,\Box\, P_2) = last'(P_1)$. This rule puts P' in the control, and we know by the induction hypothesis that $last'(P_1) = last'(P')$ and, thus, the lemma holds.

(Sequential Composition 1) This case is analogous to (External Choice 1 and 2).

(Sequential Composition 2) $last'(P; Q) = last'(Q)$. Therefore the lemma holds trivially by the definition of $last'$.

(Synchronized Parallelism 1, 2 and 3) It is the same case as (External Choice 1 and 2).

(Hiding 1 and 2) This case is similar to (External Choice 1 and 2). Note that the loss of the specification position (it is $\bullet$ in R') is only used as a flag to add nodes to the graph; but it does not have any influence on the derivation.

(Renaming 1) It is completely analogous to the previous case.

■

Lemma 4. *Let $\mathcal{S}$ be a CSP specification, $\mathcal{D}$ a complete derivation of $\mathcal{S}$ performed with the tracking semantics, and $\mathcal{G}$ the graph produced by $\mathcal{D}$. Then, for each sequential composition $(P; Q)$ in the control of the left state of a rewriting step in $\mathcal{D}$, we have that $last'(P)$ and $\mathcal{P}os(;)$ are nodes of $\mathcal{G}$ and $\mathcal{P}os(;)$ is the successor of all the elements of the set $last'(P)$ whenever P has successfully finished.*

Proof. If a sequential composition $(P; Q)$ is in the left state of the control of a rewriting step, following the tracking semantics, only the rules (Sequential Composition 1) and (Sequential Composition 2) can be applied. (Sequential Composition 1) is only used to evolve process P until it is finished. The application of this rule is only possible with any event except ✓, remaining the sequential composition operator in the control. (Sequential Composition 2) can only be used when ✓ happens and thus Ω is left in the control.

Therefore, when P has successfully finished evolving to Ω and with n as the new reference, (Sequential Composition 2) is applied. This rule adds to the graph a new node n labeled with the specification position of ; that has successor p (a fresh reference). Therefore, we have that $\mathcal{P}os(;)$ is a node of $\mathcal{G}$ and $\mathcal{P}os(;)$ is the successor of the node m. Then, we have to prove that $last'(P)$ is a set of nodes of the graph added before P successfully finished with reference n and its successor is p.

We prove this claim by induction on the length of the derivation $P \overset{\Theta_0}{\rightsquigarrow} \ldots \overset{\Theta_n}{\rightsquigarrow} \Omega$, $n \geq 0$. The base case happens when the last rewriting step of the derivation is done leaving Ω in the control.

Only these rules can be used:

(SKIP) In this case, $m \underset{n}{\mapsto}$ `SKIP` is added to $\mathcal{G}$ and n is the new reference. Because $last'(\texttt{SKIP}) = \{\texttt{SKIP}\}$, therefore, the claim follows.

(External Choice 3) Here, $last'(P_1 \,\square\, P_2) = last'(P_1)$. This rule puts P' in the control which is Ω by the conditions of the lemma. Therefore, there must be at least one `SKIP`, which is $last'(P)$, at the top of Θ_n because we know that the derivation successfully finishes and thus Θ_n is finite.

(External Choice 4) It is analogous to the previous case, but here $last'(P_1 \,\square\, P_2) = last'(P_2)$.

(Synchronized Parallelism 4) $last'(P_1 \mid\mid P_2) = last'(P_1) \cup last'(P_2)$. The parents of the last nodes of P_1 and P_2 are connected to the new reference r. Therefore the claim follows.

(Hiding 3) In this case, because $last'(P\backslash B) = last'(P)$ and P is put in the control of the left state in the rewriting step of the precondition which must be reduced to Ω performing $\checkmark$, the claim follows by the recursive application of one of these six rules.

(Renaming 2) It is completely analogous to the previous case.

The induction hypothesis states that for all rewriting step $R \overset{\Theta}{\rightsquigarrow} R', R' \neq \Omega$ in the derivation $Q \overset{\Theta_0}{\rightsquigarrow} \ldots \overset{\Theta_n}{\rightsquigarrow} \Omega, n \geq 0$ where $P \overset{\Theta'}{\rightsquigarrow} Q \in \mathcal{D}$, $last'(P)$ is put in the control of a further rewriting step of the derivation together with its reference.

Then, we prove that this also holds for the previous rewriting step $R_0 \overset{\Theta''}{\rightsquigarrow} R$. Only the rules that do not perform $\checkmark$ could be applied (because $\checkmark$ puts Ω in the control of the right state and now, we are not considering the final rewriting step).

(STOP) This rule could not be applied because it puts $\not\Omega$ in the control. There is no rule for $\not\Omega$ thus, if applied, P could not successfully finished.

(Process Call), (Prefixing), (Internal Choice 1 and 2) and (Conditional Choice 1 and 2) In these rules R is put in the control of the final state together with its reference. We know by Lemma 3 that $last'(R_0) = last'(R)$ thus, the claim follows by the induction hypothesis.

(External Choice 1 and 2) Both rules keep the process in the control and the same references, thus the claim follows by the induction hypothesis.

(External Choice 3 and 4) In this case, $last'(P_1 \,\square\, P_2) = last'(P_1)$. This rule puts P' in the control, and we know by Lemma 3 that $last'(P_1) = last'(P')$, thus the lemma holds by the induction hypothesis.

(Sequential Composition 1 and 2) We know that P successfully finished, thus (Sequential Composition 1) is applied a number of times before (Sequential Composition 2), that puts Q in the control. We know that $last'(P;Q) = last'(Q)$ thus, the claim holds by the induction hypothesis.

(Synchronized Parallelism 4) $last'(P_1||P_2) = last'(P_1) \cup last'(P_2)$. The parents of the last nodes of P_1 and P_2 are connected to the new reference r. Therefore the claim follows.

(Hiding 3) In this case, because $last'(P\backslash B) = last'(P)$ and P is put in the control of the condition state which must be reduced to Ω performing $\checkmark$, the claim follows by the recursive application of one of these rules.

(Renaming 2) It is completely analogous to the previous case.

■

In the following lemma, for the sake of clarity, we use an extended notion of rewriting step where each state is represented with a pair that includes the process to be evaluated and the reference to the current node in the graph. *Extended* rewriting steps are represented with $(P, m) \overset{\Theta}{\rightsquigarrow} (P', m')$.

Lemma 5. *Let $\mathcal{S}$ be a CSP specification, $\mathcal{D}$ a complete derivation of $\mathcal{S}$ performed with the tracking semantics, and $\mathcal{G}$ the graph produced by $\mathcal{D}$. Then, for each extended rewriting step in $\mathcal{D}$ of the form $(P, m) \overset{\Theta}{\rightsquigarrow} (P', m')$ which is not associated with* (Synchronized Parallelism 4) *we have that a node for first(P) is added to $\mathcal{G}$ with reference m.*

Proof. We prove the lemma for each rule.

(SKIP), (STOP), (Prefixing), (Process Call), (Parameterized Process Call), (Internal Choice 1 and 2) and (Conditional Choice 1 and 2) A node for $\mathit{first}(P)$ (in these rules, α) is added to $\mathcal{G}$ with reference m.

(External Choice 1, 2, 3 and 4) and (Synchronized Parallelism 1, 2 and 3) In these rules, the node associated with $\mathit{first}(P)$ (α here) could be included or not, depending on whether it has been included by another rewriting step. If it is already included, it is due to the specification position of the previous expression in the control is the same as P, and its associated rewriting step or another one has added it. Otherwise, function `FirstEval` is called and it includes the node for P with reference m, since n is equal to $\bullet$.

(Hiding 1, 2 and 3) and (Renaming 1 and 2) The proof for these rules is similar to the one for rules of external choice and synchronized parallelism, but here function $\texttt{FirstEval}_{\texttt{HR}}(\mathcal{G}, (\alpha, \backslash B), m)$ is called and it adds to $\mathcal{G}$ the node for P with reference m because the label α is distinct from $\bullet$.

(Sequential Composition 1 and 2) In both rules, the node for $\mathit{first}(P)$ is included by a rewriting step in Θ. All possible rewriting steps must apply one of these previous rules, and thus, the claim recursively follows.

■

Lemma 6. *Let $\mathcal{S}$ be a CSP specification, $\mathcal{D}$ a complete derivation of $\mathcal{S}$ performed with the tracking semantics, and $\mathcal{G}$ the graph produced by $\mathcal{D}$. Then, for each rewriting step in $\mathcal{D}$ of the form $R_i \overset{\Theta_i}{\rightsquigarrow} R_{i+1}$ we have that:*

1. *E_c contains an arc $\mathcal{P}os(R_i) \mapsto \mathcal{P}os(\mathit{first}(R'))$ where $R' \overset{\Theta'}{\rightsquigarrow} R'' \in \Theta_i$ and $R_i \Rightarrow \mathit{first}(R')$, and*
2. *if $R_i \Rightarrow \mathit{first}(R_{i+1})$ then E_c contains an arc $\mathcal{P}os(R_i) \mapsto \mathcal{P}os(\mathit{first}(R_{i+1}))$.*

Proof. We prove each claim separately:

1. Firstly, we know that Θ cannot be empty. Therefore, rules (SKIP), (STOP), (Process Call), (Prefixing), (Internal Choice 1 and 2), (Conditional Choice 1 and 2) and (Synchronized Parallelism 4) could not be applied. Moreover, (Sequential Composition) could never be applied because if R_i is of the form $P;Q$, then the unique possible case is that $\mathcal{P}os(P;Q) \Rightarrow \mathcal{P}os(Q)$ (by Definition 2). And Q can only be in the control of the right state; hence, Q cannot appear in Θ. Then the only applicable rules are (External Choice) or (Synchronized Parallelism).

   Let us consider extended rewriting steps $(R', m') \overset{\Theta'}{\rightsquigarrow} (R'', m'') \in \Theta_i$. First, we have to prove that a node with the specification position of R_i is included in the graph and the reference of its successor node is put in each m' of Θ_i. In rules (External Choice) and (Synchronized Parallelism) it is done using function `FirstEval`. The references associated with the selected branches of the operator must be $\bullet$, i.e., the branches have not been developed until now in the derivation. Otherwise, by Definition 2, there is no possible control flow between R_i and R'. In this case, if the corresponding reference is $\bullet$, then `FirstEval` adds to $\mathcal{G}$ the specification position of R_i and the reference of the successor node is put in all possible m'.

2. In this case, R_i cannot be neither a `SKIP` nor a `STOP`, because $\mathcal{P}os(R_i) \not\Rightarrow \mathcal{P}os(R_{i+1})$ (Ω or $\not\Omega$, respectively) by Definition 2. Process R_i cannot be a parallelism because $\mathcal{P}os(R_i) \not\Rightarrow \mathcal{P}os(R_{i+1})$ (itself or Ω).

   If R_i is an external choice we have two possibilities. If we apply (External Choice 1 or 2) then R_i and R_{i+1} have the same specification position and thus, by Definition 2, no control flow is possible. If we apply (External Choice 3 or 4) the control cannot pass from R_i to R_{i+1}, because R_{i+1} is different to $first(R_i.1)$ or $first(R_i.2)$. This is due to the fact that the nodes associated with these positions have necessarily been added to $\mathcal{G}$ by the rewriting step Θ_i or by a previous rewriting step on derivation $\mathcal{D}$. Therefore, process R_i must be a process call, a prefixing, an internal choice, a conditional choice or a sequential composition. If it is a sequential composition, rule (Sequential Composition 1) cannot be applied because in this case R_i and R_{i+1} have the same specification position. Therefore, only (Sequential Composition 2) can be applied.

We now prove that the application of any of remaining rules (Process Call), (Prefixing), (Internal Choice 1 and 2), (Conditional Choice 1 and 2), and (Sequential Composition 2) satisfies the property.

Let $(R_i, n_i) \overset{\Theta_i}{\rightsquigarrow} (R_{i+1}, n_{i+1})$ be an extended rewriting step. In all the rules, a node labeled α is added to $\mathcal{G}$ (except in (Prefixing) where is β) and the position of its successor is placed as n_{i+1}. Furthermore, we know by Lemma 5 that a node for $\mathcal{P}os(first(R_{i+1}))$ is included in the next rewriting step in the derivation $(R_{i+1}, n_{i+1}) \overset{\Theta_{i+1}}{\rightsquigarrow} (R_{i+2}, n_{i+2})$ having associated position n_{i+1}. Note here again that Lemma 5 excludes rule (Synchronized Parallelism 4) but in this case both branches must be already in $\mathcal{G}$ by a previous application of (Synchronized Parallelism 1, 2 or 3). ■

Lemma 7. *Let $\mathcal{S}$ be a CSP specification, $\mathcal{D}$ a derivation of $\mathcal{S}$ performed with the tracking semantics, and $\mathcal{G}$ the graph produced by $\mathcal{D}$. Then, there exists a synchronization arc ($a \leftrightarrow a'$) in $\mathcal{G}$ for each synchronization in $\mathcal{D}$ where a and a' are the nodes of the synchronized events.*

Proof. We prove this lemma by induction on the length of the derivation $\mathcal{D} = R_0 \overset{\Theta_0}{\rightsquigarrow} R_1 \overset{\Theta_1}{\rightsquigarrow} \ldots \overset{\Theta_n}{\rightsquigarrow} R_{n+1}$. We can assume that the derivation starts with the initial configuration $(\texttt{MAIN}_{(\texttt{MAIN},0)}, \emptyset, 0)$, thus in the base case, the only rule applicable is (Process Call) or (Parameterized Process Call) and hence no synchronization is possible. We assume as the induction hypothesis that there exists a synchronization arc $(a \leftrightarrow a') \in \mathcal{G}$ for each synchronization in $R_0 \overset{\Theta_0}{\rightsquigarrow} \ldots \overset{\Theta_{i-1}}{\rightsquigarrow} R_i$ with $0 < i \leq n$ and prove that the lemma also holds for the next rewriting step $R_i \overset{\Theta_i}{\rightsquigarrow} R_{i+1}$.

Firstly, only (Synchronized Parallelism 3) allows the synchronization of events. Therefore, only if R_i is a synchronizing parallelism, or if a (Synchronized Parallelism 3) is applied in Θ_i, $(a \leftrightarrow a') \in \mathcal{G}$. Then, let us consider the case where $\overset{\Theta_i}{\rightsquigarrow}$ is the application of rule (Synchronized Parallelism 3). This proof is also valid in the case where (Synchronized Parallelism 3) is applied in Θ_i. We have the following rewriting step:

$$\frac{\mathit{RewritingStep}_1 \qquad\qquad \mathit{RewritingStep}_2}{(P_1 \underset{X}{\|}_{(\alpha,n_1,n_2)} P_2, \mathcal{G}, m) \xrightarrow{(a,\Delta_1\cup\Delta_2)} (P_1' \underset{X}{\|}_{(\alpha,n_1'',n_2'')} P_2', \mathcal{G}'', m)}$$

$$\text{with } a \in X$$

$$\text{and where } \mathcal{G}'' = \mathcal{G}_1'' \cup \mathcal{G}_2'' \cup \{s_1 \overset{a}{\leftrightarrow} s_2 \mid s_1 \in \Delta_1 \wedge s_2 \in \Delta_2\}$$

$$\begin{aligned}
&\wedge\ \mathit{RewritingStep}_1 = (P_1, \mathcal{G}_1', n_1') \xrightarrow{(a,\Delta_1)} (P_1', \mathcal{G}_1'', n_1'') \\
&\wedge\ (\mathcal{G}_1', n_1') \ = \ \texttt{FirstEval}(\mathcal{G}, n_1, m, (\alpha, \underset{X}{\|})) \\
&\wedge\ \mathit{RewritingStep}_2 = (P_2, \mathcal{G}_2', n_2') \xrightarrow{(a,\Delta_2)} (P_2', \mathcal{G}_2'', n_2'') \\
&\wedge\ (\mathcal{G}_2', n_2') \ = \ \texttt{FirstEval}(\mathcal{G}, n_2, m, (\alpha, \underset{X}{\|}))
\end{aligned}$$

Because (Prefixing) is the only rule that performs an event a without further conditions, we know that P_1 must be a prefixing operator or a process containing a prefixing operator whose prefix is a, i.e., we know that the rule applied in $\mathit{RewritingStep}_1$ is fired with an event a; and we know that all the rules of the semantics except (Prefixing) need to fire another rule with an event a as a condition. Therefore, at the top of the condition rules, there must be a (Prefixing). The same happens with P_2. Hence, two prefixing rules (one for P_1 and one for P_2) have been fired as a condition of this rule.

In addition, the new graph $\mathcal{G}''$ contains the synchronization set $\{s_1 \overset{a}{\leftrightarrow} s_2 \mid s_1 \in \Delta_1 \wedge s_2 \in \Delta_2\}$ where Δ_1 and Δ_2 are the sets of references to the events that must synchronize in $\mathit{RewritingStep}_1$ and $\mathit{RewritingStep}_2$, respectively.

Hence, we have to prove that all and only the references to event a (that must synchronize in $\mathit{RewritingStep}_1$) are in Δ_1. We prove this by showing that all references to the synchronized events are propagated down by all rules from the (Prefixing) in the top to the (Synchronized Parallelism 3). The proof is analogous for $\mathit{RewritingStep}_2$.

The possible rules applied in $(P_1, \mathcal{G}'_1, n') \xrightarrow{(a,\Delta_1)} (P'_1, \mathcal{G}''_1, n''_1)$ are:

(Prefixing) In this case, the set Δ_1 only contains the reference to event a.

(Synchronized Parallelism 3) In this case, the sets Δ_1 and Δ_2 are joined and propagated down.

(External Choice 3 and 4), (Sequential Composition 1), (Synchronized Parallelism 1 and 2), (Hiding 1 and 2), (Renaming 1) In these cases, the set Δ is propagated down.

Therefore, all the synchronized events are in the set Δ_1 and the claim follows. ■

Theorem 2 (Semantics correctness). *Let $\mathcal{S}$ be a CSP specification, $\mathcal{D}$ a derivation of $\mathcal{S}$ performed with the tracking semantics, and $\mathcal{G}$ the graph produced by $\mathcal{D}$. Then, $\mathcal{G}$ is the track associated with $\mathcal{D}$.*

Proof. In order to prove that $\mathcal{G} = (N, E_c, E_s)$ is a track, we need to prove that it satisfies the properties of Definition 5. For each $R \overset{\Theta}{\rightsquigarrow} R' \in \mathcal{D}$ and for all rewriting steps in Θ we have

1. E_c contains a control-flow arc $a \mapsto a'$ iff $a \Rrightarrow a'$ with respect to $\mathcal{D}$. This is ensured by the three clauses of Definition 4:
   - by Lemma 2, if R is a prefixing $(a \to P)$, then E_c contains an arc $\mathcal{P}os(a) \mapsto \mathcal{P}os(\to)$;
   - by Lemma 4, if R is a sequential composition $(Q; P)$, then E_c contains an arc $\forall p \in \mathit{last}'(Q)$, $\mathcal{P}os(p) \mapsto \mathcal{P}os(;)$;
   - by Lemma 6, if $R \Rightarrow \mathit{first}(R'')$ where $R'' \overset{\Theta'}{\rightsquigarrow} R''' \in \Theta$, then E_c contains an arc $\mathcal{P}os(R) \mapsto \mathcal{P}os(\mathit{first}(R''))$; and if $R \Rightarrow \mathit{first}(R')$ then E_c contains an arc $\mathcal{P}os(R) \mapsto \mathcal{P}os(\mathit{first}(R'))$; and
2. by Lemma 7, E_s contains a synchronization arc $a \leftrightarrow a'$ for each synchronization occurring in the rewriting step where a and a' are the synchronized events.

Moreover, we know that the only nodes in N are the nodes induced by E_c and E_s because all the nodes inserted in $\mathcal{G}$ are inserted by connecting the new node to the last inserted node (i.e., if the current reference is m and the new fresh reference is n, then the new node is always inserted as $\mathcal{G}[m \underset{n}{\mapsto} \alpha]$). Hence, all nodes are related by control or synchronization arcs and thus the claim holds. ■

Our last result states that the trace of a derivation can be extracted from its associated track. To prove it, we define first an order on the event nodes of a track that corresponds to the order in which they were generated by the tracking semantics.

Definition 7. *(Event node order) Given a track $\mathcal{G} = (N, E_c, E_s)$ and nodes $m, n \in N$ such that $l(m), l(n) \in \Sigma$, m is* smaller *than n, represented by $m \ll n$ iff $m' < n'$ where $(m, m'), (n, n') \in E_c$.*

Intuitively, an event node m is smaller than an event node n if and only if the successor of m has a reference smaller than the reference of the successor of n. The following lemma is also necessary to prove that the order in which events occur in a derivation is directly related with the order of Definition 7. In the following we consider an augmented version of derivation $\mathcal{D}$ which includes the event fired by the application of the rule. So, we can represent derivation $\mathcal{D}$ as $P_1 \underset{e_1}{\overset{\Theta_1}{\rightsquigarrow}} \ldots \underset{e_j}{\overset{\Theta_j}{\rightsquigarrow}} P_{j+1}$.

Lemma 1. *Given a derivation $\mathcal{D} = P_1 \underset{e_1}{\overset{\Theta_1}{\rightsquigarrow}} \ldots \underset{e_j}{\overset{\Theta_j}{\rightsquigarrow}} P_{j+1}$ of the tracking semantics, and the track $\mathcal{G} = (N, E_c, E_s)$ produced by $\mathcal{D}$, then $\forall e_i \in \Sigma, 1 \leq i \leq j$,*

- *$\exists n \in N$ such that $l(n) = e_i$, and*
- *$\exists (n, n') \in E_c$ such that $n' = n + 1$.*

Proof. In order to prove this lemma, we prove first that any rewriting step $P_i \overset{\Theta_i}{\underset{e_i}{\rightsquigarrow}} P_{i+1}$ in $\mathcal{D}$, $1 \leq i \leq j$, with $e_i \in \Sigma$ is a prefixing or it performs a prefixing in Θ_i. This can be easily proved by showing that the rewriting step is either a prefixing (thus $\Theta_i = \emptyset$), or Θ_i has a prefixing as the top rewriting step. We prove it by induction on the length of Θ_i.
In the base case, $\Theta_i = \emptyset$. This case happens when the rule applied is a prefixing, thus the claim follows trivially. We assume as the induction hypothesis that the claim follows for a Θ_i with a depth of n. And we prove the claim for a depth of $n + 1$ by case analysis. The only possible rules applied in the rewriting step at depth n of Θ_i are:

(External Choice 3 and 4), (Synchronized Parallelism 1 and 2), (Sequential Composition 1) and (Hiding 2) In these rules the nth rewriting step of Θ_i contains a single rewriting step whose event is also e_i. Moreover, this rewriting step (the nth+1 rewriting step of Θ_i) must be a prefixing, because it is the only applicable rule with an event e_i and with a $\Theta = \emptyset$. Hence, the claim follows.

(Synchronized Parallelism 3) In this case, the nth+1 rewriting step of Θ_i is formed by two different rewriting steps, and both of them must be a prefixing whose associated event is being synchronized. Otherwise, the depth of Θ_i would be greater than n+1 and, thus, the claim also follows in this case.

Now, both conditions hold trivially from the fact that the prefixing rule adds n to N with label $l(n) = e_i = \alpha$, and it also adds the prefixing operator ($\rightarrow$) to N as the successor of n. ■

Therefore, Lemma 1 ensures that the order of Definition 7 corresponds to the order in which the semantics generates the nodes, because each event is added to the graph together with a new fresh reference for the prefixing operator. Since references are generated incrementally, the occurrence of an event e will generate a reference which is less than the reference generated with a posterior event e'. With this order, we can easily define a transformation to extract a trace from a track based on the following proposition:

Proposition 1. *Given a track $\mathcal{G} = (N, E_c, E_s)$, the trace induced by $\mathcal{G}$ is the sequence of events $T = e_1, \ldots, e_m$ that labels the associated sequence of nodes $T' = n_1, \ldots, n_m$ (i.e., $\forall e_i \in T, n_i \in T'$, $1 \leq i \leq m, l(n_i) = e_i$ and $e_i \in \Sigma$) where:*

1. *$\forall n_i \in T',\ 0 < i < m,\ n_i \ll n_{i+1}$.*
2. *$\forall n \in N$ such that $l(n) \in \Sigma$, if $(\nexists n' \in N \mid (n, n') \in E_s)$, then $n \in T'$.*
3. *$\forall n \in N$ such that $l(n) \in \Sigma$, if $(\forall n' \in N \mid (n, n') \in E_s \wedge\ n' \ll n)$, then $n \in T'$.*

Proof. We consider a derivation $\mathcal{D} = P_1 \overset{\Theta_1}{\underset{e'_1}{\rightsquigarrow}} \ldots \overset{\Theta_j}{\underset{e'_k}{\rightsquigarrow}} P_{j+1}$. Note that $e'_1, \ldots, e'_k \neq e_1, \ldots, e_m$ because the former contains events in $\{\tau, \checkmark\}$. Then, we have that the trace is the subsequence of $e'_1, \ldots, e'_k$ that only includes events of Σ. We will represent this subsequence with $\mathcal{E} = e'_j, \ldots, e'_{j'}$ with $0 \leq j \leq j' \leq k$. Then, we have to show that $T = \mathcal{E}$. The proposition follows trivially from the fact that the sequence T follows the order of nodes imposed by Definition 7, and this order is the same order of the events that form the sequence $\mathcal{E}$ as stated by Lemma 1. ■

Theorem 3 (Track correctness). *Let $\mathcal{S}$ be a CSP specification, $\mathcal{D}$ a derivation of $\mathcal{S}$ produced by the sequence of events (i.e., the trace) $T = e_1, \ldots, e_m$, and $\mathcal{G}$ the track associated with $\mathcal{D}$. Then, there exists a function f that extracts the trace T from the track $\mathcal{G}$, i.e., $f(\mathcal{G}) = T$.*

Proof. Proposition 1 allows to trivially define a function f such that $f(\mathcal{G}) = T$ being $\mathcal{G}$ the track of a derivation $\mathcal{D}$, and being T the trace of the same derivation. For a track $\mathcal{G} = (N, E_c, E_s)$ we have that

$$f((n : ns), E_c, E_s) = \begin{cases} f((ns), E_c, E_s) & \text{if } (\exists n' \in N | (n, n') \in E_s \wedge n \ll n') \\ (l(n) : f((ns), E_c, E_s)) & \text{otherwise} \end{cases}$$

where list $(n : ns)$ corresponds to the set $\{n \in N \mid l(n) \in \Sigma\}$ ordered with respect to order $\ll$ of Definition 7. ■