A Tracking Semantics for CSP*

Marisa Llorens, Javier Oliver, Josep Silva, and Salvador Tamarit

Universidad Politécnica de Valencia,
Camino de Vera S/N, E-46022 Valencia, Spain
{mllorens,fjoliver,jsilva,stamarit}@dsic.upv.es

Abstract. CSP is a powerful language for specifying complex concurrent systems. Due to the non-deterministic execution order of processes and to synchronizations, many analyses such as deadlock analysis, reliability analysis, and program slicing try to predict properties of the specification which can guarantee the quality of the final system. These analyses often rely on the use of CSP's traces. In this work, we introduce the theoretical basis for tracking concurrent and explicitly synchronized computations in process algebras such as CSP. Tracking computations 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. Formal definition of a tracking semantics improves the understanding of the tracking process, but also, it allows to formally prove the correctness of the computed tracks.

Key words: Concurrent Programming, CSP, Semantics, Tracking.

1 Introduction

One of the most important techniques for program understanding and debugging is tracing [3]. A trace gives the user access to otherwise invisible 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; and thus, a tracer is a powerful tool to explore, understand and debug concurrent computations.

One of the most widespread concurrent specification languages is the *Communicating Sequential Processes* (CSP) [7, 17] whose operational semantics allows the combination of parallel, non-deterministic and non-terminating processes.

* This work has been partially supported by the Spanish *Ministerio de Ciencia e Innovación* under grant TIN2008-06622-C03-02, by the *Generalitat Valenciana* under grant ACOMP/2009/017, and by the *Universidad Politécnica de Valencia* (Programs PAID-05-08 and PAID-06-08). Salvador Tamarit was partially supported by the Spanish MICINN under FPI grant BES-2009-015019.

The study and transformation of CSP specifications often uses different analyses such as deadlock analysis [10], reliability analysis [8] and program slicing [19] which are based on a data structure able to represent computations.

In CSP a trace is a sequence of events. Concretely, the operational semantics of CSP is an event-based semantics in which the occurrence of events fires the rules of the semantics. Hence, the final trace of the computation is the sequence of events occurred (see Chapter 8 of [17] for a detailed study of this kind of traces). In this work we introduce an essentially different notion of trace [3] called track. In our setting, a track is a data structure which represents the sequence of expressions that have been evaluated during the computation, and moreover, this data structure is labelled 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 following CSP specification:[1]

$$\texttt{MAIN} = \texttt{CASINO} \parallel \texttt{GAMBLING}$$

$$\texttt{CASINO} = (\texttt{PLAYER} \mid\mid\mid \texttt{ROULETTE}) \underset{\{\texttt{betred},\texttt{red},\texttt{black},\texttt{prize}\}}{\parallel} \texttt{CROUPIER}$$

$$\texttt{PLAYER} = \texttt{betred} \rightarrow (\texttt{prize} \rightarrow \texttt{STOP} \;\Box\; \texttt{noprize} \rightarrow \texttt{STOP})$$

$$\texttt{ROULETTE} = \texttt{red} \rightarrow \texttt{STOP} \;\Box\; \texttt{black} \rightarrow \texttt{STOP}$$

$$\begin{aligned}\texttt{CROUPIER} = {} & (\texttt{betred} \rightarrow \texttt{red} \rightarrow \texttt{prize} \rightarrow \texttt{STOP}) \\ & \Box\ (\texttt{betred} \rightarrow \texttt{black} \rightarrow \texttt{prize} \rightarrow \texttt{STOP}) \\ & \Box\ (\texttt{betblack} \rightarrow \texttt{black} \rightarrow \texttt{prize} \rightarrow \texttt{STOP}) \\ & \Box\ (\texttt{betblack} \rightarrow \texttt{red} \rightarrow \texttt{getmoney} \rightarrow \texttt{STOP})\end{aligned}$$

$$\texttt{GAMBLING} = \textit{Complex Composite Processes}$$

This specification models several gambling activities running in parallel and modelled by process GAMBLING. One of the games is the casino. A CASINO is modelled as the interaction of three parallel processes, namely a PLAYER, a ROULETTE, and a CROUPIER. The player bets for red, and she can win a prize or not. The roulette simply takes a color (either red or black); and the croupier checks the bet and the color of the roulette in order to give a prize to the player or just get the bet money.

This specification contains an error, because it allows the trace of events $t = \langle \texttt{betred}, \texttt{black}, \texttt{prize} \rangle$ where the player bets for red and she wins a prize even though the roulette takes black.

[1] We refer those readers non familiar with CSP syntax to Section 2 where we provide a brief introduction to CSP.

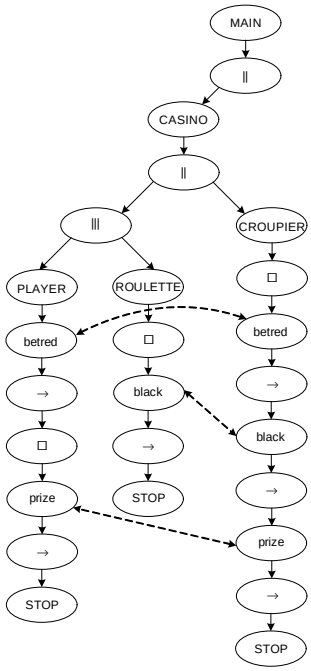


Fig. 1. Track of the program in Example 1

Now assume that we execute the specification and discover the error after executing trace t. A track can be very useful to understand why the error was caused, and what part of the specification was involved in the wrong execution. For instance, if we look at the track of Fig. 1, we can easily see that the three processes run in parallel, and that the prize is given because there is a synchronization (dashed edges represent synchronizations) between `CROUPIER` and `PLAYER` that should never happen. Observe that the track is intuitive enough as to be a powerful program comprehension tool that provides much more information than the trace.

Moreover, observe that the track contains explicit information about the specification expressions that were involved in the execution. Therefore, it can be used for program slicing (see [18] for an explanation of the technique and [14] for an adaptation of program slicing to CSP). In particular, in this example, we can use the track to extract the part of the program that was involved in the execution—note that this is the only part that could cause the error—. This part has been underscored in the example. With a quick look, one can see that the underscored part of process `CROUPIER` produced the wrong behavior. Event `prize` should be replaced by `getmoney`.

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 [14] for an example and an explanation of this transformation). In our example, even though the system is very big due to the process `GAMBLING`, the track is able to extract the only information related to the trace.

We have implemented a tool [13] able to produce tracks and to automatically color parts of the code related to some point in the specification. This tool is integrated in the last version of ProB [11, 12] which is the most extended IDE for CSP.

In languages such as Haskell, the tracks (see, e.g., [3–5, 1]) are the basis of many analysis methods and tools. However, computing CSP tracks is a complex task due to the non-deterministic execution of processes, due to deadlocks, due to non-terminating processes and mainly due to synchronizations. This is probably the reason why no correctness result exists which formally relates the track of a specification to its execution. This semantics is needed because it would 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 is only one attempt to define and build tracks for CSP [2]. Their notion of track is based on the standard *program dependence graph* [6]; therefore it is useful for program slicing but it is insufficient for other analyses that need a *context-sensitive graph* [9] (i.e., each different process call has a different representation). Moreover, their notion of track does not include synchronizations. Our tracks are able to represent synchronizations, and they are context-sensitive.

The main contributions of this work are the formal definition of tracks, the definition of the first tracking semantics for CSP and the proof that the trace of a computation can be extracted from the track of this computation. 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. However, we design the semantics in such a way that the track is produced incrementally step by step. Therefore, if the execution is stopped (e.g., by the user because it is non-terminating or because a limit in the size of the track was specified), 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. Firstly, in Section 2 we recall the syntax and semantics of CSP. In Section 3 we define the concept of track for CSP. Then, in Section 4, 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 5, 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. Finally, Section 6 concludes.

2 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 2 summarizes the syntax constructions used in CSP specifications. A *specification* is viewed as a finite set of process definitions. The left-hand side of each definition is the name of a process, which is defined in the right-hand side (abbrev. *rhs*) by means of an expression that can be a call to another process or a combination of the following operators:

Prefixing It specifies that event a must happen before process P.

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

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

Sequential composition It specifies a sequence of two processes. When the first (successfully) finishes, the second starts.

Synchronized parallelism Both processes are executed in parallel with a set X of synchronized events. In absence of synchronizations both processes can execute in any order. Whenever a synchronized event $a \in X$ 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., $X = \emptyset$).

Skip It successfully finishes the current process. It allows us to continue the next sequential process.

Stop Synonymous with deadlock: It finishes the current process and it does not allow the next sequential process to continue.

$S ::= \{D_1, \ldots, D_m\}$	(Entire specification)	*Domains* $M, N \ldots \in \mathcal{N}$ (Process names) $P, Q \ldots \in \mathcal{P}$ (Processes) $a, b \ldots \in \Sigma$ (Events)
$D ::= N = P$	(Process definition)	
$P ::= M$	(Process call)	
$\mid\ a \rightarrow P$	(Prefixing)	
$\mid\ P \sqcap Q$	(Internal choice)	
$\mid\ P \,\Box\, Q$	(External choice)	
$\mid\ P \,;\, Q$	(Sequential composition)	
$\mid\ P \underset{X}{\parallel} Q$	(Synchronized parallelism)	where $X \subseteq \Sigma$
$\mid\ SKIP$	(Skip)	
$\mid\ STOP$	(Stop)	

Fig. 2. Syntax of CSP specifications

We now recall the standard operational semantics of CSP as defined by Roscoe [17]. It is presented in Fig. 3 as a logical inference system. A *state* of the semantics is a process to be evaluated called the *control*. The 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 $\Sigma = \{a, b, c \ldots\}$ 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 Fig. 3. The intuitive meaning of each rule is the following:

(Process Call) The call is unfolded and the right-hand side of process named N is added to the control.

(Prefixing) When event a occurs, process P is added to the control.

(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 added to the control.

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

(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 starts. Note that $\checkmark$ is hidden from outside the whole process becoming τ.

(Synchronized Parallelism 1 and 2) When event $a \notin X$ or events τ or $\checkmark$ happen, one of the two processes P or Q evolves accordingly, but only a is visible from outside the parallelism operator.

(Synchronized Parallelism 3) When 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, performing the event $\checkmark$.

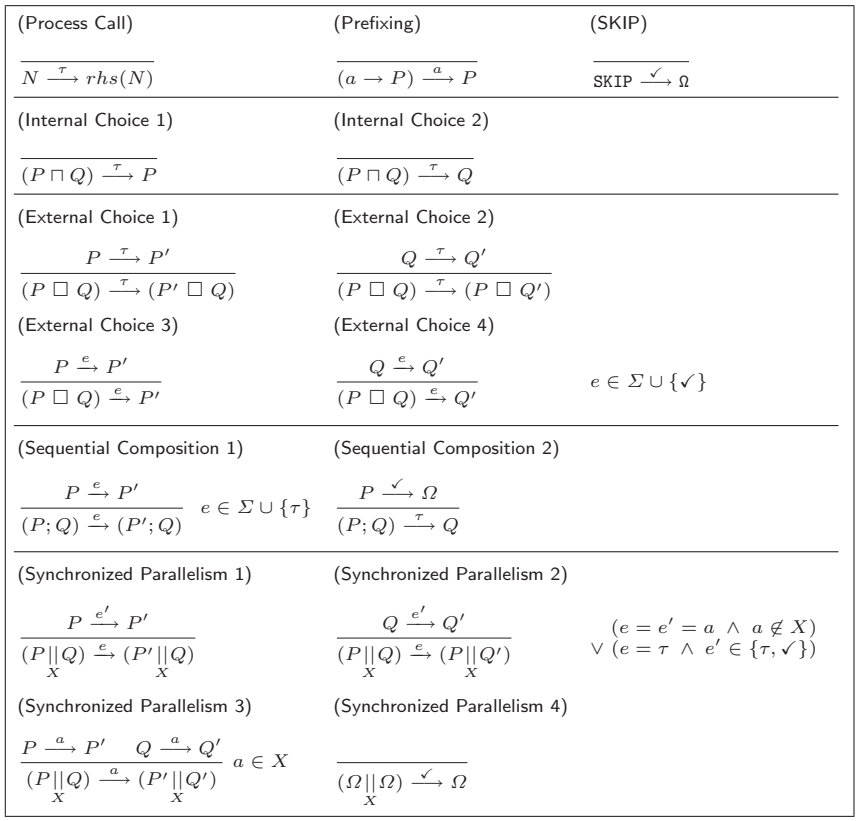


Fig. 3. CSP's operational semantics

We illustrate the semantics with the following example.

Example 2. Consider the next CSP specification:

$$\texttt{MAIN} \;=\; (\texttt{a} \rightarrow\; \texttt{STOP}) \underset{\{\texttt{a}\}}{\parallel} (\texttt{P} \;\Box\; (\texttt{a} \rightarrow\; \texttt{STOP}))$$

$$\texttt{P} \;=\; \texttt{b} \rightarrow\; \texttt{SKIP}$$

If we use `MAIN` as the initial state to execute the semantics, we get the computation shown in Fig. 4 where the final state is $((\texttt{a} \rightarrow \texttt{STOP}) \underset{\{\texttt{a}\}}{\parallel} \Omega)$. This computation corresponds to the execution of the left branch of the choice (i.e., `P`) and thus only event `b` occurs. Each rewriting step is labelled with the applied rule, and the example should be read from top to bottom.

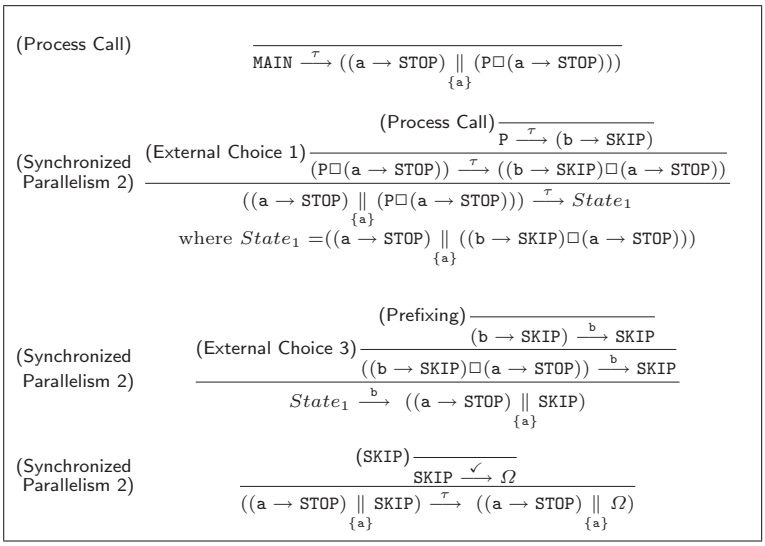


Fig. 4. A computation with the operational semantics in Fig. 3

3 Tracking Computations

In this section we define the notion of 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. The inclusion of source positions in the track implies an additional level of complexity in the semantics, but the benefits of providing our 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 which 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 and it is defined over nodes of an abstract syntax tree for a CSP specification. Formally,

Definition 1. *(Specification position) A* specification position *is a pair* (N, w) *where* $N \in \mathcal{N}$ *and* w *is a sequence of natural numbers (we use* Λ *to denote the empty sequence). We let* $\mathcal{P}os(o)$ *denote the specification position of an expression* o*. Each process definition* $N = P$ *of a CSP specification is labelled with specification positions. The specification position of its left-hand side is* $\mathcal{P}os(N) = (N, 0)$*. The right-hand side is labelled with the call* `AddSpPos`$(P, (N, \Lambda))$*; where function*

`AddSpPos` *is defined as follows:*

$$\texttt{AddSpPos}(P,(N,w)) = \begin{cases} P_{(N,w)} & \textit{if } P \in \mathcal{N} \\ STOP_{(N,w)} & \textit{if } P = STOP \\ SKIP_{(N,w)} & \textit{if } P = SKIP \\ a_{(N,w.1)} \rightarrow_{(N,w)} \texttt{AddSpPos}(Q,(N,w.2)) & \textit{if } P = a \rightarrow Q \\ \texttt{AddSpPos}(Q,(N,w.1))\ op_{(N,w)}\ \texttt{AddSpPos}(R,(N,w.2)) & \textit{if } P = Q\ op\ R\ \ \forall\ op \in \{\sqcap,\Box,||,;\} \end{cases}$$

Example 3. Consider again the CSP specification in Example 2 where literals are labelled with their associated specification positions (they are underlined) so that labels are unique:

$$\texttt{MAIN}_{\underline{(\texttt{MAIN},0)}} = (\texttt{a}_{\underline{(\texttt{MAIN},1.1)}} \rightarrow_{\underline{(\texttt{MAIN},1)}} \texttt{STOP}_{\underline{(\texttt{MAIN},1.2)}}) \underset{\{\texttt{a}\}}{\parallel}_{\underline{(\texttt{MAIN},\Lambda)}} (\texttt{P}_{\underline{(\texttt{MAIN},2.1)}} \Box_{\underline{(\texttt{MAIN},2)}} (\texttt{a}_{\underline{(\texttt{MAIN},2.2.1)}} \rightarrow_{\underline{(\texttt{MAIN},2.2)}} \texttt{STOP}_{\underline{(\texttt{MAIN},2.2.2)}}))$$

$$\texttt{P}_{\underline{(\texttt{P},0)}} = \texttt{b}_{\underline{(\texttt{P},1)}} \rightarrow_{\underline{(\texttt{P},\Lambda)}} \texttt{SKIP}_{\underline{(\texttt{P},2)}}$$

In the following, specification positions will be represented with greek letters ($\alpha, \beta, \ldots$) and we will often use indistinguishably an expression and its associated specification position when it is clear from the context (e.g., in Example 3 we will refer to $(\texttt{P},\texttt{1})$ 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 individual literals of a CSP specification are executed. Intuitively, the control 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) [18] of imperative programming. 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 $N = \mathit{rhs}(N) \in \mathcal{S}$
ii) $\alpha \in \{\sqcap,\Box,||\}\ \wedge\ \beta \in \{\mathit{first}(\alpha.1), \mathit{first}(\alpha.2)\}$
iii) $\alpha \in \{\rightarrow,;\}\ \wedge\ \beta = \mathit{first}(\alpha.2)$
iv) $\alpha = \beta.1\ \wedge\ \beta = \rightarrow$
v) $\alpha \in \mathit{last}(\beta.1)\ \wedge\ \beta = ;$

where $first(\alpha)$ is the specification position of the subprocess denoted by α which must be executed first:

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

and $last(\alpha)$ is the set of all possible termination points of the subprocess denoted by α:

$$last(\alpha) = \begin{cases} \{\alpha\} & \text{if } \alpha = \texttt{SKIP} \\ \emptyset & \text{if } \alpha = \texttt{STOP} \vee \\ & (\alpha \in \{||\} \wedge (last(\alpha.1) = \emptyset \vee last(\alpha.2) = \emptyset)) \\ last(\alpha.1) \cup last(\alpha.2) & \text{if } \alpha \in \{\sqcap, \Box\} \vee \\ & (\alpha \in \{||\} \wedge last(\alpha.1) \neq \emptyset \wedge last(\alpha.2) \neq \emptyset) \\ last(\alpha.2) & \text{if } \alpha \in \{\rightarrow, ;\} \\ last((N, \Lambda)) & \text{if } \alpha = N \end{cases}$$

For instance, in Example 3, we can see how the control can pass from a specification position to another one, e.g., we have $(\texttt{MAIN}, 2) \Rightarrow (\texttt{MAIN}, 2.1)$ and $(\texttt{MAIN}, 2) \Rightarrow (\texttt{MAIN}, 2.2.1)$ due to rule ii). And $(\texttt{MAIN}, 2.2.1) \Rightarrow (\texttt{MAIN}, 2.2)$ due to rule iv); $(\texttt{MAIN}, 2.2) \Rightarrow (\texttt{MAIN}, 2.2.2)$ due to rule iii) and $(\texttt{MAIN}, 2.1) \Rightarrow (\texttt{P}, 1)$ due to rule i).

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 $P \overset{\Theta}{\leadsto} P'$, is the transformation of P into P' by using a rule of the CSP semantics. Therefore, $P \overset{\Theta}{\leadsto} P'$ iff a rule of the form $\dfrac{\Theta}{P \xrightarrow{e} P'}$ is applicable, where $e \in \Sigma \cup \{\tau, \checkmark\}$ and Θ is a (possibly empty) set of rewriting steps. Given a CSP process P_0, we say that the sequence $P_0 \overset{\Theta_0}{\leadsto} \ldots \overset{\Theta_n}{\leadsto} P_{n+1}$, $n \geq 0$, is a* derivation *of P_0 iff $\forall\ i, 0 \leq i \leq n, P_i \overset{\Theta_i}{\leadsto} P_{i+1}$ 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, in Fig. 5(a), one (possible) complete derivation of Example 3 is shown (for the time being, the reader can ignore the underlined part). The rules applied in each rewriting step (ignoring subderivations) are (Process Call) and (Synchronized Parallelism 3) (abbrev. (PC) 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 derivation only reaches (non-deterministically) a set of them. Therefore, we will use in the following a modified version of *last* called $last'$ whose behaviour is exactly the same as *last* except in the case of choices where only one of the branches is selected:

For each derivation $(P \sqcap P' \overset{\Theta}{\leadsto} P)$ or $(P \Box P' \overset{\Theta_0}{\leadsto} \ldots \overset{\Theta_n}{\leadsto} P'', n \geq 0$ such that $P \overset{\Theta'_0}{\leadsto} \ldots \overset{\Theta'_m}{\leadsto} P'', m \geq 0)$, $last'(P \sqcap P') = last'(P \Box P') = last'(P)$.

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. 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 ($a \rightarrow Q$) or a sequential composition ($Q; R$), then $\mathcal{P}os(a) \Rrightarrow \mathcal{P}os(\rightarrow)$ 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 $\frac{}{(a \rightarrow P) \xrightarrow{a} P}$. In this case, clause 1 guarantees that the control can dynamically pass from a to $\rightarrow$; and clause 3 guarantees that the control can dynamically pass from $\rightarrow$ to P. However, in general, Θ is not empty, and the rewriting step is of the form $\frac{P'' \longrightarrow P'''}{P \longrightarrow P'}$. 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 that we have a rewriting step to evaluate the process $P \,\Box\, P'$. Clearly, the control can pass from $\Box$ to both P and P' ($\Box \Rightarrow P$ and $\Box \Rightarrow P'$), but in the rewriting step the control will only pass to one of them ($\Box \Rrightarrow P$ or $\Box \Rrightarrow P'$). 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.

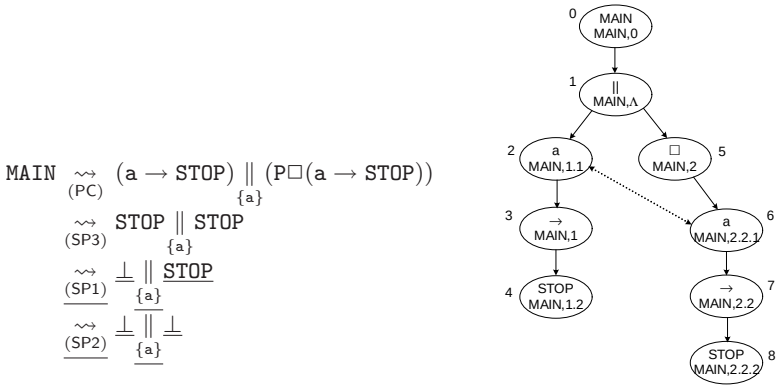
Definition 5. *(Track) Given a CSP specification $\mathcal{S}$, and a derivation $\mathcal{D}$ in $\mathcal{S}$, the* track *of $\mathcal{D}$ is a graph $\mathcal{G} = (N, E_c, E_s)$ where N is a set of nodes uniquely identified with a natural number and that are labelled with specification positions ($l(n)$ refers to the* label *of node n), and edges are divided into two groups:*

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

and

1. *E_c contains a control-flow edge $a \mapsto a'$ iff $a \Rrightarrow a'$ with respect to $\mathcal{D}$, and*
2. *E_s contains a synchronization edge $a \leftrightarrow a'$ 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.



(a) Derivation (b) Track

Fig. 5. Derivation and track associated with the specification of Example 3

Example 4. Consider again the specification of Example 3. We show in Fig. 5(a) one possible derivation (ignoring subderivations) of this specification (for the time being, the underlined part should be ignored). Its associated track is shown in Fig. 5(b). In the example, we see that the track is a connected and directed graph. Apart from the control-flow edges, there is one synchronization edge between nodes $(\texttt{MAIN}, 1.1)$ and $(\texttt{MAIN}, 2.2.1)$ representing the synchronization of event $\texttt{a}$. To illustrate the inclusion of edges in Definition 5, we see that the edge between nodes 2 and 3 is introduced according to clause 1 of Definition 4; the edge between nodes 5 and 6 is introduced according to clause 2 of Definition 4 because, in the subderivations of (SP3), there is a rewriting step $\dfrac{\dfrac{}{(\texttt{a} \to \texttt{STOP}) \xrightarrow{\texttt{a}} (\texttt{STOP})}\,\text{(Prefixing)}}{(\texttt{P}\Box(\texttt{a} \to \texttt{STOP})) \xrightarrow{\texttt{a}} (\texttt{STOP})}\,\text{(External Choice 4)}$ and $\mathit{first}(\texttt{a} \to \texttt{STOP}) = \texttt{a}$; the edge between nodes 7 and 8 is introduced according to clause 3 of Definition 4 because there is also a rewriting step $\dfrac{}{(\texttt{a} \to \texttt{STOP}) \xrightarrow{\texttt{a}} (\texttt{STOP})}\,\text{(Prefixing)}$ and $\mathit{first}(\texttt{STOP}) = \texttt{STOP}$; and the synchronization edge between nodes 2 and 6 is introduced according to clause 2 of Definition 5.

The trace associated with the derivation in Fig. 5(a) is $\langle \texttt{a} \rangle$. 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.

4 Instrumenting the Semantics for Tracking

The generation of tracks in CSP introduces new challenges such as non-deterministic execution of processes, deadlocks, non-terminating processes and synchronizations. In this work, we design a solution that overcomes these difficulties. Firstly, we generate tracks with an augmented semantics which is conservative with respect to the standard operational semantics. Therefore, the execution order is the standard order, thus non-determinism and synchronizations are solved by the semantics. 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 5. In the following CSP specification two non-terminating processes run in parallel and synchronize infinitely.

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

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

Because the computation is infinite, the track (shown in Fig. 6) is also infinite.

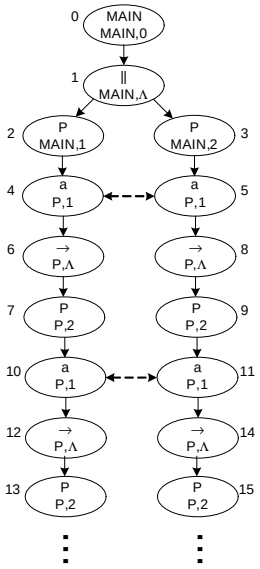


Fig. 6. Track of the program in Example 5

In order to solve the problem of deadlocks (that stop the computation), and have a representation for them in the tracks; when a deadlock happens, the semantics performs some additional steps to be able to generate a part of the track that represents the deadlock. These additional steps do 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 Fig. 7, where we assume that every literal in the program has been labelled with its specification position (denoted by a subscript, e.g., P_α). In this semantics, a *state* is a tuple (P, G, m, Δ), where P is the process to be evaluated (the *control*), G is a directed graph (i.e., the track built so far), m is a numeric reference to the current node in G, and Δ is a set of references to nodes that may be synchronized. Concretely, m references the node in G where the specification position of the control P must be stored. Reference m is a fresh[2] reference generated to add new nodes to 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 $G[m \underset{n}{\mapsto} \alpha]$ to introduce a node in G. For instance, if we are adding a node to G this new node has reference m, it is labelled with specification position α, and its successor is n (a fresh reference). Successor arrows are denoted by $m \underset{n}{\mapsto}$ which means that node n is the successor of node m. Every time an event in Σ happens during the computation, this event 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 $\bot$ to denote any process that is deadlocked. In order to perform computations, we construct an initial state (e.g., $(\texttt{MAIN}, \emptyset, 0, \emptyset)$) and (non-deterministically) apply the rules of Fig. 7. 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:

(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 $\emptyset$.

(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. The set Δ is $\{m\}$ to indicate that event a has occurred and it 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 $\bot$ (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.

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

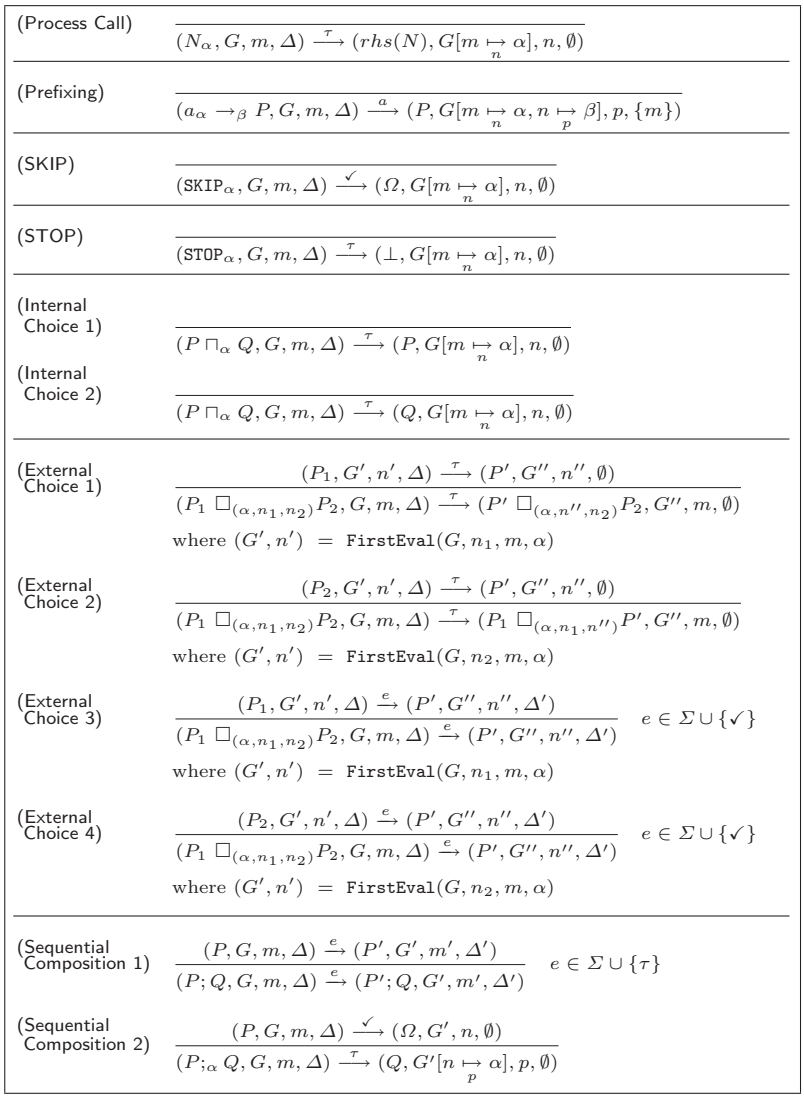


Fig. 7. An instrumented operational semantics to generate CSP tracks

(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) External choices can 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 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 labelling choice operators with a

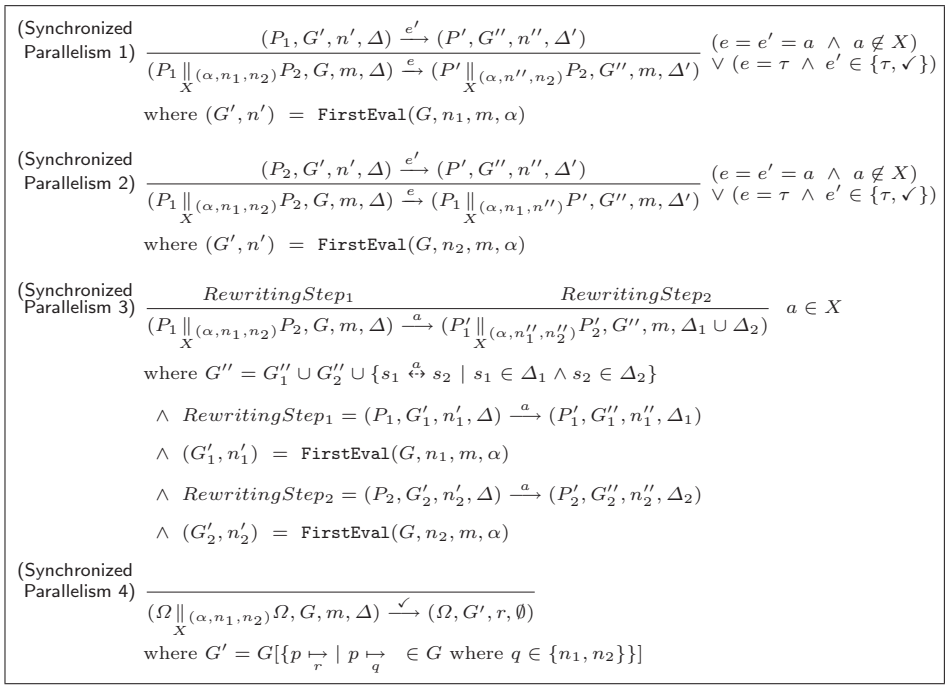


Fig. 7. An instrumented operational semantics to generate CSP tracks (cont.)

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:

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

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 G. For instance, consider the rewriting step (EC4) of Fig. 8. The choice operator in the rewriting step R is labelled with $((\texttt{MAIN}, \Lambda), \bullet, \bullet)$. Therefore, it is evaluated for the first time, and thus, in the left-hand side state of the upper rewriting step, node $5 \underset{6}{\mapsto} (\texttt{MAIN}, 2)$, which refers to the choice operator, is added to G.

(Sequential Composition 1 and 2) Sequential Composition 1 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 compo-

sition 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 labelled 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 it happens in the external choice rules. For instance, consider the rewriting step (Synchronized Parallelism 3) of Fig. 8. The parallelism operator in the rewriting step *State 1* is labelled 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 L, node $1 \underset{2}{\mapsto} (\texttt{MAIN}, \Lambda)$, which refers to the parallelism operator, is added to G.

(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 (and synchronized) event. Each branch derivation has a non-empty set of events (Δ_1, Δ_2) to be synchronized (note that this is a set because many parallelisms could be nested). Then, all references in the sets Δ_1 and Δ_2 are mutually linked with synchronization edges. Both sets are 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 edge 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.

We illustrate this semantics with a simple example.[3]

Example 6. Consider again the specification in Example 3. Figure 5(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 $(\texttt{MAIN}, \emptyset, 0, \emptyset)$ shown in Fig. 8. Here, for clarity, each computation step is labelled with the applied rule; in each state, G denotes the current graph. This computation corresponds to the execution of the right branch of the choice (i.e., $\texttt{a} \rightarrow \texttt{STOP}$). The final state is $(\bot \underset{\{\texttt{a}\}}{\parallel}_{((\texttt{MAIN},\Lambda),9,10)} \bot, G', 1, \emptyset)$. The final track G' computed for this execution

[3] We refer the reader to [16] where another example is discussed.

is depicted in Fig. 5(b) where we can see that nodes are numbered with the references generated by the instrumented semantics. Note that nodes 9 and 10 were prepared by the semantics (edges to them were produced) but never used because the subcomputations were stopped in STOP. Note also that the track contains all the parts of the specification executed by the semantics. This means that if the left branch of the choice had been developed (i.e., unfolding the call to P, thus using rule (External Choice 3)), this branch would also belong to the track.

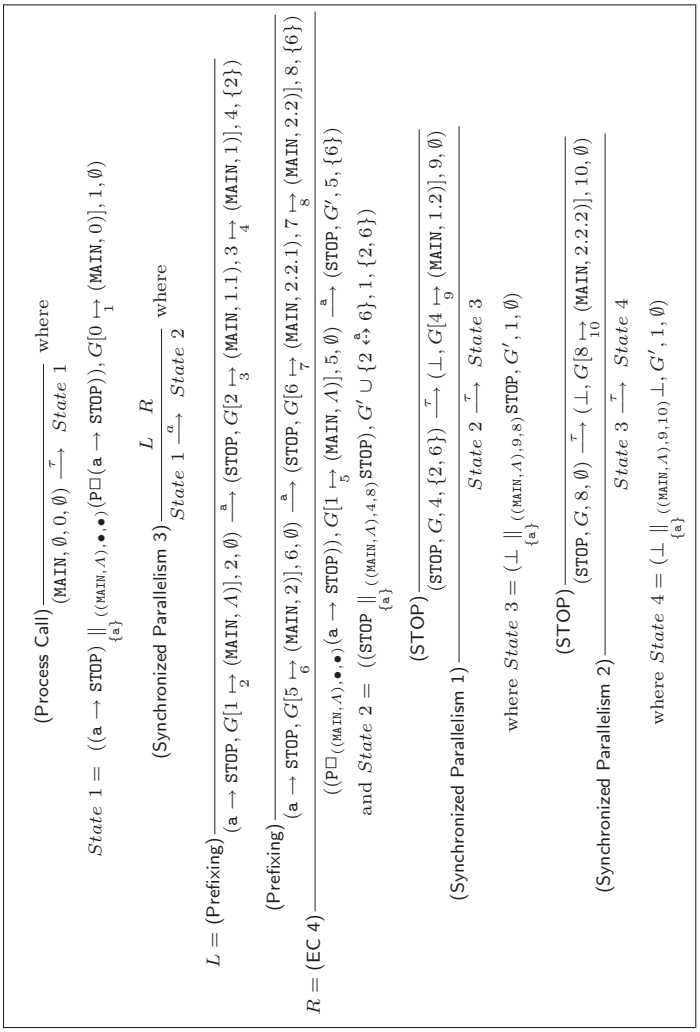


Fig. 8. An example of computation with the tracking semantics in Fig. 7

5 Correctness

In this section we prove the correctness of the tracking semantics (in Fig. 7) 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 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 finishes, while the standard semantics finishes without performing any additional step.

Theorem 1 (Conservativeness). *Let $\mathcal{S}$ be a CSP specification, P a process in $\mathcal{S}$, and $\mathcal{D}$ and $\mathcal{D}'$ the derivations of P performed with the standard semantics of CSP and with the tracking semantics, respectively. Then, the sequence of rules applied in $\mathcal{D}$ and $\mathcal{D}'$ is exactly the same except that $\mathcal{D}'$ performs one rewriting step more than $\mathcal{D}$ 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 $\bot$ in the control:

$$\overline{(\texttt{STOP}_\alpha, G, m, \Delta) \xrightarrow{\tau} (\bot, G[m \underset{n}{\mapsto} \alpha], n, \emptyset)}$$

Then, the (sub)computation is stopped because no rule is applicable for $\bot$. 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` that perform 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 (proven in [16]) are used.

Lemma 1. *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 $(a \rightarrow P)$ in the control of the left state of a rewriting step in $\mathcal{D}$, we have that $\mathcal{P}os(a)$ and $\mathcal{P}os(\rightarrow)$ are nodes of $\mathcal{G}$ and $\mathcal{P}os(\rightarrow)$ is the successor of $\mathcal{P}os(a)$.*

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 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.*

Lemma 3. *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 edge $\mathcal{P}os(R_i) \mapsto \mathcal{P}os(first(R'))$ where $R' \overset{\Theta'}{\rightsquigarrow} R'' \in \Theta_i$ and $R_i \Rightarrow first(R')$, and*
2. *if $R_i \Rightarrow first(R_{i+1})$ then E_c contains an edge $\mathcal{P}os(R_i) \mapsto \mathcal{P}os(first(R_{i+1}))$.*

Lemma 4. *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 edge $(a \nleftrightarrow a')$ in $\mathcal{G}$ for each synchronization in $\mathcal{D}$ where a and a' are the nodes of the synchronized events.*

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 edge $a \mapsto a'$ iff $a \Rrightarrow a'$ with respect to $\mathcal{D}$. This is ensured by the three clauses of Definition 4:
   - by Lemma 1, if R is a prefixing $(a \rightarrow P)$, then E_c contains an edge $\mathcal{P}os(a) \mapsto \mathcal{P}os(\rightarrow)$;
   - by Lemma 2, if R is a sequential composition $(Q;P)$, then E_c contains an edge $\forall p \in last'(Q), \mathcal{P}os(p) \mapsto \mathcal{P}os(;)$;
   - by Lemma 3, if $R \Rightarrow first(R'')$ where $R'' \overset{\Theta'}{\rightsquigarrow} R''' \in \Theta$, then E_c contains an edge $\mathcal{P}os(R) \mapsto \mathcal{P}os(first(R''))$; and if $R \Rightarrow first(R')$ then E_c contains an edge $\mathcal{P}os(R) \mapsto \mathcal{P}os(first(R'))$; and
2. by Lemma 4, E_s contains a synchronization edge $a \nleftrightarrow a'$ 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 $G[m \underset{n}{\mapsto} \alpha]$). Hence, all nodes are related by control or synchronization edges 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 6. *(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 6. 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 $P_1 \overset{\Theta_1}{\underset{e_1}{\rightsquigarrow}} \ldots \overset{\Theta_j}{\underset{e_j}{\rightsquigarrow}} P_{j+1}$.

Lemma 5. *Given a derivation* $\mathcal{D} = P_1 \overset{\Theta_1}{\underset{e_1}{\rightsquigarrow}} \ldots \overset{\Theta_j}{\underset{e_j}{\rightsquigarrow}} P_{j+1}$ *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 5 (proven in [16]) ensures that the order of Definition 6 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* $n \in T'$

The proof of this proposition can be found in [16].

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 6.

6 Conclusions

This work introduces the first semantics of CSP instrumented for tracking. Therefore, it is an interesting result because it can serve as a reference mark to define and prove properties such as completeness of static analyses which are based on tracks [13–15]. The execution of the tracking semantics produces a graph as a side effect which is the track of the computation. This track is produced 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. The generated track can be useful not only for tracking computations but for debugging and program comprehension. This is due to the fact that our generated track also includes the specification positions associated with the expressions appearing in the track. Therefore, tracks could be used to analyse what parts of the program are executed (and in what order) in a particular computation. Also, this information allows a track viewer tool to highlight the parts of the code that are executed in each step. Notable analyses that use tracks are [3–5, 1, 13–15]. The introduction of this semantics allows us to adapt these analyses to CSP. On the practical side, we have implemented a tool called *SOC* [13] which is able to automatically generate tracks of a CSP specification. These tracks are later used for debugging. *SOC* has been integrated into the most extended CSP animator and model-checker ProB [11, 12], that shows the maturity and usefulness of this tool and of tracks. The implementation, source code and several examples are publicly available at: `http://users.dsic.upv.es/~jsilva/soc/`

Acknowledgements

We want to thank the anonymous referees for many valuable comments and useful suggestions.

References

1. 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)

2. 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)
3. 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)
4. 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)
5. Chitil, O., Lou, Y.: Structure and Properties of Traces for Functional Programs. Electronic Notes in Theoretical Computer Science (ENTCS). 176(1), 39–63 (2007)
6. 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)
7. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Upper Saddle River, NJ, USA (1985)
8. 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)
9. Krinke, J.: Context-Sensitive Slicing of Concurrent Programs. ACM SIGSOFT Software Engineering Notes. 28(5) (2003)
10. Ladkin, P., Simons, B.: Static Deadlock Analysis for CSP-Type Communications. Responsive Computer Systems (Chapter 5), Kluwer Academic Publishers (1995)
11. 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)
12. Leuschel, M., Fontaine, M.: Probing the depths of CSP-M: A new FDR-compliant validation tool. In: Liu, S., Maibaum, T., Araki, K. (eds.) ICFEM 2008. LNCS, vol. 5256, pp. 278–297. Springer, Heildeberg (2008)
13. 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)
14. Leuschel, M., Llorens, M., Oliver, J., Silva, J., Tamarit, S.: The MEB and CEB Static Analysis for CSP Specifications. In: Hanus, M. (ed.) LOPSTR 2008, Revised Selected Papers. LNCS, vol. 5438, pp. 103–118. Springer, Heildeberg (2009)
15. Llorens, M., Oliver, J., Silva, J., Tamarit, S.: An Algorithm to Generate the Context-sensitive Synchronized Control Flow Graph. In: 25th ACM Symposium on Applied Computing (SAC 2010) (to appear). ACM, New York, NY, USA (2010)
16. Llorens, M., Oliver, J., Silva, J., Tamarit, S.: A Tracking Semantics for CSP (Extended Version). Technical report, DSIC-II/03/10, Universidad Politécnica de Valencia (March 2010)
17. Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice Hall, Upper Saddle River, NJ, USA (2005)
18. Tip, F.: A survey of program slicing techniques. Journal of Programming Languages. 3, 121–189 (1995)
19. Weiser, M.D.: Program Slicing. IEEE Transactions on Software Engineering. 10(4), 352–357 (1984)