Reversible CSP Computations

Carlos Galindo, Naoki Nishida, Josep Silva and Salvador Tamarit

***Abstract*—Reversibility enables a program to be executed both forwards and backwards. This ability allows programmers to backtrack the execution to a previous state. This is essential if the computation is not deterministic because re-running the program forwards may not lead to that state of interest. Reversibility of sequential programs has been well studied and a strong theoretical basis exists. Contrarily, reversibility of concurrent programs is still very young, especially in the practical side. For instance, in the particular case of the Communicating Sequential Processes (CSP) language, reversibility is practically missing. In this work, we present a new technique, including its formal definition and its implementation, to reverse CSP computations. Most of the ideas presented can be directly applied to other concurrent specification languages such as Promela or CCS, but we center the discussion and the implementation on CSP. The technique proposes different forms of reversibility, including strict reversibility and causal-consistent reversibility. On the practical side, we provide an implementation of a system to reverse CSP computations that is able to highlight the source code that is being executed in each forwards/backwards computation step, and that has been optimized to be scalable to real systems.**

***Index Terms*—Concurrent programming, tracing, debugging aids, code inspections and walkthroughs.**

I. Introduction

ONE of the most extended languages for the specification of concurrent processes is *Communicating Sequential Processes* (CSP) [1], [2]. The specification, analysis, and transformation of CSP specifications have often been based on the use of *CSP traces* [2], which represent the sequences of events that can occur in a system. In fact, there are different analyses such as deadlock analysis [3], [4], livelock analysis [5], and security analysis [6], among others, that are based on or use this kind of traces.

Example 1.1: Consider the following CSP specification:[1]

```
channel a,b
MAIN = P ||{a} Q
P = a → b → SKIP
Q = b → a → ((b → SKIP) □ Q)
```

This work has been partially supported by the EU (FEDER) and the Spanish MCI/AEI under grants TIN2016-76843-C4-1-R and PID2019-104735RB-C41, by the *Generalitat Valenciana* under grant Prometeo/2019/098 (DeepTrust), by JSPS KAKENHI under Grant Number JP17H01722, and by TAILOR, a project funded by EU Horizon 2020 research and innovation programme under GA No 952215. (*Corresponding author: Josep Silva*).

C. Galindo and J. Silva are with the Departamento de Sistemas Informáticos y Computadores, Universitat Politècnica de València, Camino de Vera s/n, 46022 Valencia, Spain (e-mail: jsilva@dsic.upv.es).

N. Nishida is with Graduate School of Informatics, Nagoya University, Furo-cho, Chikusa-ku, 464-8603 Nagoya, Japan.

S. Tamarit is with PFS Tech, Avda. Cortes Valencianas 58 E, 46015 Valencia, Spain.

[1]Readers that are not familiarized with the CSP syntax are referred to Section II, where a gentle introduction to CSP is given.

The only possible traces of this specification are {⟨⟩,⟨b⟩,⟨ba⟩,⟨bab⟩,⟨babb⟩}. If we consider the trace ⟨babb⟩, it can be produced by four different computations due to the non-deterministic evaluation order of the processes. While the first two events (ba) are deterministic (i.e., the b event corresponds to the first b in process Q, and the a event corresponds to the synchronization of the a events in P and Q), the last two b events in the trace can be produced by P and one of the two branches of Q interleavedly.

Unfortunately, standard CSP traces are not adequate for those analyses that need to relate the trace with the source code (e.g., debugging). The trace of a computation does not always provide enough information to identify a bug in the source code, or even to decide whether the computation is buggy. For instance, if we execute the program in Example 1.1 and we get the trace ⟨babb⟩, we cannot know what parts of the code have been executed. It could be the case that process Q is buggy (e.g., it should be Q = b → a → (c → SKIP □ Q)). But, even if the buggy code of Q (i.e., b instead of c) was actually executed, we would not know it, because the trace produced by the buggy code is identical to a correct trace (following another path). The problem still remains if the generated trace is wrong. For instance, if the last event of the trace (b) is buggy we cannot know what part of the program produced this buggy event (in fact, it could be any of the three b events in the specification). And, what is even more important, if we are interested in a particular buggy event, we cannot trace the error backward (reversing the computation [7]). A proper reversibility tool for CSP would allow programmers to undo unproductive computations, and to backtrack to a save state when an error is detected.

We overcome these problems in this work. Concretely, we propose an operational semantics of CSP that is (i) conservative (i.e., the forward evaluation follows the standard operational semantics of CSP), and that (ii) produces as a side effect the *history of the computation* such that every evaluated expression has information associated with the source code. In this way, (1) we can know exactly what expressions of the program are evaluated at each step (i.e., we can associate the events that occur in the computation with the corresponding literals in the source code), and (2) the history of the computation allows us to reverse the computation. The proposed model has been implemented in a system to animate CSP computations forward and backward.

II. Preliminary definitions and notation

In this section we introduce some notation, and provide definitions used in the rest of the paper. Figure 1 summarizes the CSP syntax constructs that we consider in this paper. Extended explanations for each syntax construct can be found

Domains
$M, N \ldots \in \mathcal{N}$ (Process names) $P, Q \ldots \in \mathcal{P}$ (Processes)
$a, b \ldots \in \Sigma$ (Events) $u, v \ldots \in \mathcal{V}$ (Variables) $\overline{EV_n} = EV_1, \ldots, EV_n$

S	::=	$\{D_1, \ldots, D_n\}$	(Entire specification)
D	::=	$M = P$	(Process definition)
	|	$M(\overline{EV_n}) = P$	(Parameterized process)
P	::=	M	(Process call)
	|	$M(\overline{EV_n})$	(Parameterized process call)
	|	$CO \rightarrow P$	(Prefixing)
	|	$P \sqcap Q$	(Internal choice)
	|	$P \,\square\, Q$	(External choice)
	|	$P \not< Bool \not> Q$	(Conditional choice)
	|	$P \;;\; Q$	(Sequential composition)
	|	$P \underset{\{\overline{EV_n}\}}{\|\|} Q$	(Synchronized parallelism)
	|	$SKIP$	(Skip)
	|	$STOP$	(Stop)
CO	::=	$EV \mid CO?EVI \mid CO!EV$	(Compound Object)
EVI	::=	$EV \mid v : b$	(Input event with Variables)
EV	::=	$a \mid v \mid EV.EV$	(Event with Variables)
$Bool$	::=	$true \mid false \mid Bool \vee Bool$	(Boolean expression)
	|	$Bool \wedge Bool \mid \neg Bool$	
	|	$EV = EV \mid EV \neq EV$	

Fig. 1. Syntax of CSP specifications

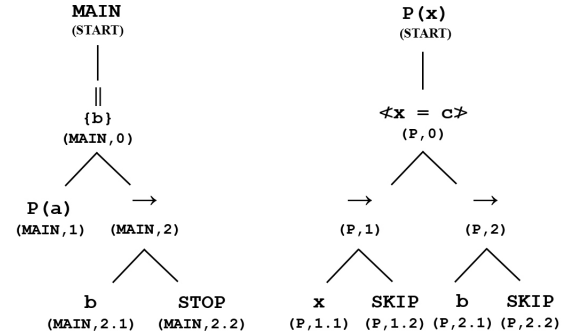

Fig. 2. Program positions of the program in Example 2.1.

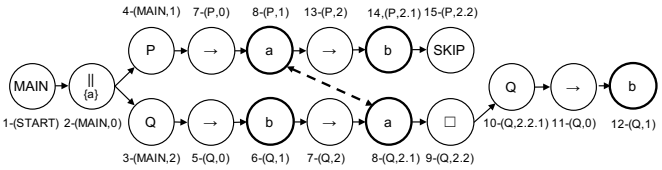

Fig. 3. R-track of a computation of the program in Example 1.1.

in [2]. A CSP *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 by means of an expression formed from the operators in Figure 1 (see Examples 1.1 and 3.11). The operational semantics of CSP is an event-based semantics, where the occurrence of events fire the rules (readers non-familiar with the standard operational semantics of CSP can find a detailed description in [2]). We use the domains Σ and Π that, respectively, contain all external (i.e., visible from the external environment) and internal events in the execution of a CSP specification.

To uniquely identify each literal in a CSP specification we use labels (that we call *program positions*). Each program position corresponds to a unique node in the CSP specification's abstract syntax tree (AST). A *program position* [8] is a pair `(P,s)` where `P` is the name of a CSP process and `s` is a sequence of natural numbers. The root of the AST is represented with `0`, and, for each operator, the operands are numbered from left to right. We use a special label `(START)` to represent the initial call to a process in the program.

Example 2.1: The following CSP program has been labelled with program positions (they are underlined):

$$\texttt{MAIN} = \texttt{P(a)}_{\underline{\texttt{(MAIN,1)}}} \underset{\{\texttt{b}\}}{\|}_{\underline{\texttt{(MAIN,0)}}} (\texttt{b}_{\underline{\texttt{(MAIN,2.1)}}} \rightarrow_{\underline{\texttt{(MAIN,2)}}} \texttt{STOP}_{\underline{\texttt{(MAIN,2.2)}}})$$
$$\texttt{P(x)} = (\texttt{x}_{\underline{\texttt{(P,1.1)}}} \rightarrow_{\underline{\texttt{(P,1)}}} \texttt{SKIP}_{\underline{\texttt{(P,1.2)}}}) \not< \texttt{x} = \texttt{c} \not> {}_{\underline{\texttt{(P,0)}}} (\texttt{b}_{\underline{\texttt{(P,2.1)}}} \rightarrow_{\underline{\texttt{(P,2)}}} \texttt{SKIP}_{\underline{\texttt{(P,2.2)}}})$$

All terms are uniquely labelled because labels keep the order of the associated AST (see Figure 2).
We define the domain $\mathcal{P}os$ to represent the set of all possible program positions. We also define $\mathcal{P}os_\Sigma \subset \mathcal{P}os$ as the subset of positions that refer to external events (i.e., events in Σ).

III. Recording the history of a CSP computation

Following the Landauer's embedding principle [9], one can make a CSP computation reversible by recording the history of the computation. In this section, we discuss about the information that must be stored to record a CSP computation and make it reversible.

We propose an extension of CSP tracks [8]—a data structure used to represent (forward) CSP computations—for storing the execution history of CSP computations, and based on it, we present a formulation and implementation of a system to reverse CSP computations.

A CSP track is a dynamic data structure that represents the sequence of expressions that have been evaluated during one computation, labelled with the location of these expressions in the specification. In contrast, a (standard) CSP trace is the sequence of events that occur during the computation [2]. Therefore, a CSP track is much more informative than a CSP trace because the former not only contains a lot of information about original program structures but it also explicitly relates the sequence of events with the parts of the specification that caused these events. Our reversibility extension of a track is called an R-track. R-tracks extend tracks with two small changes: (i) nodes are labelled with timestamps that represent the time when the expression associated with each node was produced, and (ii) the representation of prefixing is changed (in a track, `b -> SKIP` is represented with three sequential nodes: `(b)` $\rightarrow$ `(->)` $\rightarrow$ `(SKIP)`; while in an R-track it is represented as `(->)` $\rightarrow$ `(b)` $\rightarrow$ `(SKIP)`, i.e., the order of the prefix and the prefixing operator is reversed so that we can undo the prefixing operator when the event is undone).

Example 3.1: The data structure in Figure 3 is the R-track associated with a computation of the program in Example 1.1. This computation selects the recursive branch in the external choice of process `Q`. Moreover, it executes the recursive call to `Q` before executing the `b` event in process `P`. Hence, the trace produced is $\langle$`babb`$\rangle$.

Observe in Figure 3 that every single literal of the program evaluated in a computation is recorded inside a node of the

associated R-track. Moreover, each node has a label with two components t−p, where t is a timestamp that represents the instant where this node was generated. And p is the *program position* of the literal in this node. For the sake of clarity, when it is clear from the context, we often use the internal literal instead of the program position (i.e., we use (8−a) instead of (8−(P,1))).

With the timestamp we can serialize the program. For instance, if we only focus on event nodes (those with a bold line) then it is trivial to generate the associated trace ⟨babb⟩ following the sequence: (6,b)→(8,a)→(12,b)→(14,b). Observe also that the R-track records explicit information about synchronizations, which are represented with a dashed edge between the two synchronized events. Synchronizations among three or more events are represented with pairwise edges. The interested reader can find an example in the public repository of our implementation: https://github.com/tamarit/reverCSP#example. It is also important to remark that R-tracks not only record events in Σ but also internal events (i.e., $\Sigma \cup \Pi$). However, for the sake of simplicity and readability, we only consider events in Σ in this paper, because the extension to Π is straightforward (but verbose).

Once we have available a data structure such as the R-track. Three research questions emerge: (1) Can R-tracks be used to reverse a CSP computation? Yes. We discuss how to use R-tracks to reverse computations in Section III-A. (2) Can R-tracks be automatically generated from CSP computations? Yes. We have implemented a conservative extension of the standard semantics that can generate tracks efficiently from CSP computations. This is described in Section IV. (3) Is the use of R-tracks scalable to real programs? Yes. We discuss scalability issues in Section IV-A.

A. Reversing CSP computations with R-tracks

An R-track contains enough information to (re)execute the program both forward and backward in the standard way (i.e., producing a derivation of rewriting steps following the rules of the operational semantics). However, we want to go beyond the standard execution. We want to directly map the execution to the source code. So that, every single event fired along the execution is associated with a set of expressions of the source code. With this view, a derivation of a program is a sequence of pairs (e,p) where e is an event, and p is the set of program positions that point to the expressions in the source code needed to fire this event. We call such a derivation an *event–syntax trace*.

Example 3.2: Consider again the CSP program in Example 1.1. One possible event–syntax trace of this program is:

```
(b, {(START), (MAIN,0), (MAIN,2), (Q,0), (Q,1)})
→ (a, {(MAIN,1), (P,0), (P,1), (Q,2), (Q,2.1)})
→ (b, {(Q,2.2), (Q,2.2.1), (Q,0), (Q,1)})
→ (b, {(P,2), (P,2.1)})
```

This event-syntax trace is graphically represented in the second column (Running R-track) of Figure 4. There we can see that each set of grey nodes represent the portion of code that is needed to fire the associated event (represented with a bold line). With this information, we can highlight the source code that is being executed in each step, and we can do it both forward and backward.

Event–syntax traces are particularly useful for debugging, because they explicitly show what exact expressions in the source code participate in each step of the computation. Hence, one could execute the program (either forward or backward) and highlight in the source code all parts that participate in the firing of an event. To formalize the generation of event–syntax traces, we can define an R-track rewriting semantics that iteratively transforms the information of an R-track with every event of the computation. We need to provide first some formal definitions.

Definition 3.3 (R-track): An R-track is a labelled directed acyclic graph $\mathcal{G} = (N, E_c, E_s)$ where N are the nodes, 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.

Each node $n \in N$ has two labels: $time(n)$ contains a natural number (the timestamp), and $pos(n)$ contains a program position. Given two nodes $n, n' \in N$, the CSP term $pos(n)$ is executed before $pos(n')$ if and only if $time(n) < time(n')$. For the sake of clarity, we have included inside the nodes of Figures 3 and 4 a label with a source code literal. Note, however, that this label does not exist in the definition of R-track because it is actually redundant with respect to the label pos, since the program position uniquely identifies the literal. To ease the reading, we use a node n to denote the program expression that n actually represents. Hence, we use, e.g., $n \in \Sigma$ instead of $pos(n) \in \mathcal{P}os_\Sigma$, so that the notation becomes more readable. This notion of R-track is a slight conservative extension of the standard tracks. How to compute tracks from CSP specifications is described in [10], where the correctness of tracks is also proved in Theorem 3.

Because we want to execute R-tracks, we need a mechanism to know what part of the R-track has already been executed. This idea is captured with the notion of *Running R-track*.

Definition 3.4 (Running R-track): A *running R-track* is a pair $(\mathcal{G}, t)$, where $\mathcal{G} = (N, E_c, E_s)$ is an R-track and t is a natural number (a timestamp) such that $t = 0$ or $\exists n \in N \,.\, time(n) = t \ \wedge \ n \in \Sigma$.

The timestamp represents the last executed event. If the timestamp is 0, then the execution has not started yet. For instance, in the track of Figure 3, the timestamp 8 represents the occurrence of event a.

Given a running R-track $(\mathcal{G}, t)$ with $\mathcal{G} = (N, E_c, E_s)$, we define functions $\mathit{first\text{-}after}(\mathcal{G}, t)$ and $\mathit{last\text{-}before}(\mathcal{G}, t)$ to obtain the first (respectively last) event node of the R-track after (respectively before) the given timestamp t. Formally,

$$\begin{aligned}\mathit{first\text{-}after}(\mathcal{G}, t) = \{n \mid\ & t < time(n) \\ & \wedge \nexists n' \,.\, t < time(n') < time(n) \\ & \wedge n, n' \in \Sigma \wedge n, n' \in N\}\end{aligned}$$

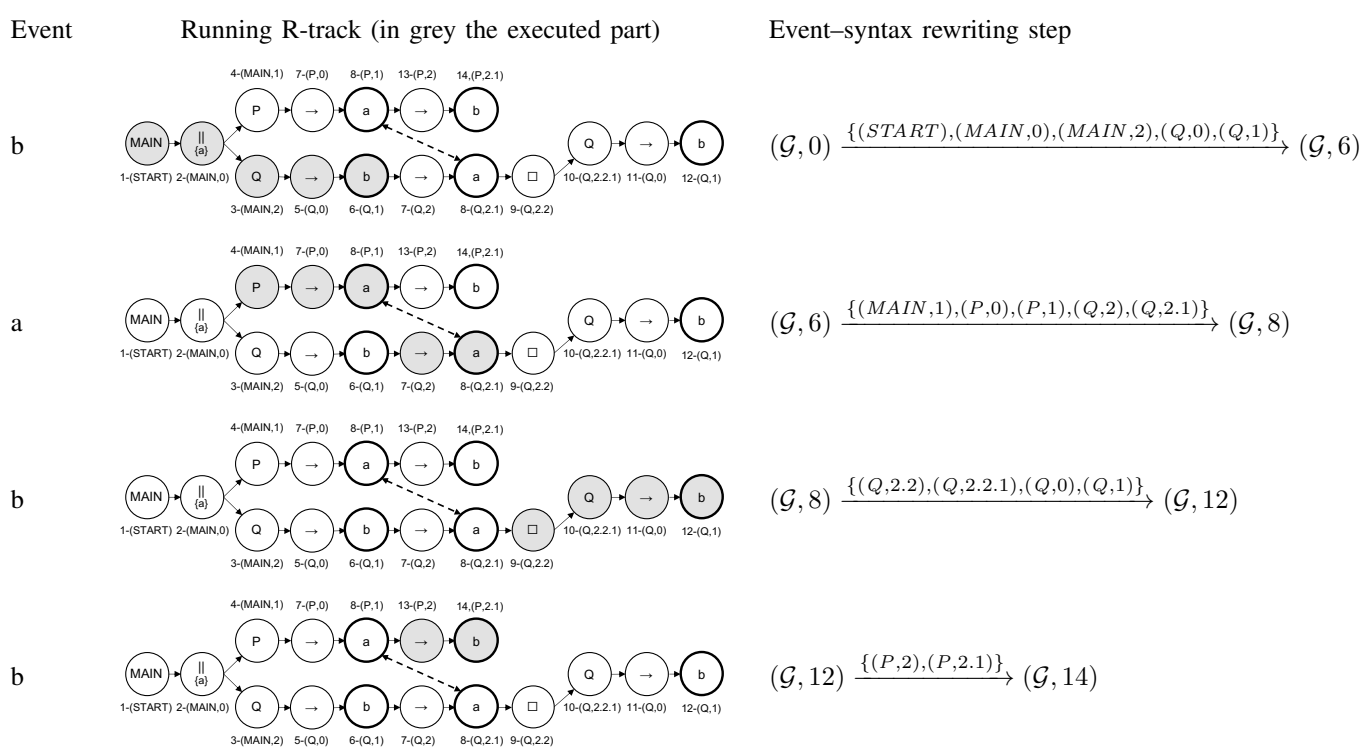

Fig. 4. Event-syntax trace associated with the trace $\langle$babb$\rangle$.

$$\begin{aligned} \mathit{last\text{-}before}(\mathcal{G},t) = \{n \mid\ & \mathit{time}(n) < t \\ & \wedge \nexists n' \,.\, \mathit{time}(n) < \mathit{time}(n') < t \\ & \wedge\, n, n' \in \Sigma \wedge n, n' \in N\} \end{aligned}$$

Note that both *first-after* and *last-before* return a set of nodes because, due to synchronizations, there could be different nodes that are the first (respectively last) nodes to be executed (all of them at the same time). A clear example of this phenomenon happens in Figure 3. The last event nodes before timestamp 14 are `8-(P,1)` and `8-(Q,2.1)` (the synchronization of `a`).

We also need to define a causality relation between nodes.

Definition 3.5 (Causality Relation in R-tracks): Given an R-track $\mathcal{G} = (N, E_c, E_s)$ and two nodes $n_1, n_2 \in N$, we say that n_1 is causal with respect to n_2 (denoted $n_1 \rightsquigarrow n_2$) if and only if $(n_1, n_2) \in E^*$ where E^* denotes the reflexive and transitive closure of E_c and E_s (E^* = ($E_c \cup E_s$)*).

The causality relation is transitive and it is induced by control-dependence and synchronizations. It is useful to determine what nodes in the R-track participate in each rewriting step. For this, we can divide the R-track into complementary sections containing a single event occurred in time, and all causal nodes of this event that are not causal with respect to other previous events (see Figure 4). This is done with function $\mathit{causalNodes}(n)$. Roughly, it returns those nodes that are causal with respect to n (i.e., they precede n in a control path) and they do not precede an event node that happened before n in the same control path. Formally, given a running R-track $(\mathcal{G}, t)$ with $\mathcal{G} = (N, E_c, E_s)$, and a node $n \in N$, we define $\mathit{causalNodes}(n)$ as:

$$\begin{aligned} \mathit{causalNodes}(\mathcal{G},n) = SYNC_n \cup \{n' \in N \mid\ & n' \notin \Sigma \wedge \\ & (n' \mapsto n_s) \in E_c^* \wedge \\ & (\nexists n'' \in \Sigma \,.\, n'' \notin SYNC_n \wedge \\ & (n' \mapsto n''), (n'' \mapsto n_s) \in E_c^*)\} \end{aligned}$$

where $n_s \in SYNC_n = \{n\} \cup \{n_{sync} \mid (n \leftrightarrow n_{sync}) \in E_s\}$, and we use E_c^* to denote the reflexive and transitive closure of E_c.

Example 3.6: Figure 4 displays four R-tracks are shown. In the first, third and fourth R-tracks, all grey nodes are the causal nodes of the last grey nodes. In the second R-track, however, not all causal nodes of the last grey node are in grey. The causal nodes of the nodes with timestamp 8 are all grey nodes plus nodes with timestamps 1 and 2.

Two nodes can share the same causal nodes. For instance, in Figure 3, nodes 1 and 2 are causal with respect to nodes 3 and 4. Given the occurrence of an event in an execution, we want to identify those nodes that caused this event *since the occurrence of the last event*. Therefore, we also need to define a function $\mathit{stepNodes}(n)$ that captures this restricted causal relation in the R-track. Roughly, it returns those nodes that precede n in the R-track and do not precede an event node that happened before n.

$$\begin{aligned} \mathit{stepNodes}(\mathcal{G},n) = \mathit{causalNodes}(\mathcal{G},n) \backslash \{n' \in N \mid \\ (n' \mapsto n'') \in E_c^* \wedge n'' \in \Sigma \wedge \mathit{time}(n'') < \mathit{time}(n)\} \end{aligned}$$

Example 3.7: In Figure 4, four R-tracks are shown. In each R-track, all grey nodes are the step nodes of the last grey nodes.

We are now in a position to define a calculus able to reproduce a computation both forward and backward from an R-track (i.e., without the need of the source code program, and without the need of the standard semantics).

Given an R-track $\mathcal{G} = (N, E_c, E_s)$, the application of the following *forward rule* is able to produce an event–syntax trace reproducing the computation forward:

$(Forward)$

$$\frac{n \in \textit{first-after}(\mathcal{G}, t) \quad t < t' = \textit{time}(n) \quad P = \{\textit{pos}(n') \mid n' \in \textit{stepNodes}(n)\}}{(\mathcal{G}, t) \xrightarrow{P} (\mathcal{G}, t')}$$

Analogously, the reverse counterpart to produce a backward derivation can be done with the application of the *backward* rule:

$(Backward)$

$$\frac{\textit{time}(n) = t > t' \quad n' \in \textit{last-before}(\mathcal{G}, t) \quad t' = \textit{time}(n') \quad P = \{\textit{pos}(n'') \mid n'' \in \textit{stepNodes}(n)\}}{(\mathcal{G}, t) \xleftarrow{P} (\mathcal{G}, t')}$$

It is important to remark that the direction of the steps of the semantics always go from left to right. The direction of the arrow only indicates whether the step is forward or backward.

Example 3.8: Let $\mathcal{G}$ be the R-track in Figure 3. The application of four rewriting steps with the *forward rule* would produce the derivation shown in the right column of Figure 4. The associated reverse computation is completely symmetric.

1) Deterministic reversibility of CSP: The technique proposed so far is a mechanism to deterministically execute forward and backward a computation strictly following the order of the original execution. That is, at any state, there is at most one possible forward execution step, and at most one possible backward execution step. Formally,

Theorem 3.9 (Deterministic forward and backward execution): Given a running R-track $(\mathcal{G}, t)$,

- $(\mathcal{G}, t) \xrightarrow{P} (\mathcal{G}, t') \wedge (\mathcal{G}, t) \xrightarrow{Q} (\mathcal{G}, t'') \implies P = Q \wedge t' = t''$
- $(\mathcal{G}, t) \xleftarrow{P} (\mathcal{G}, t') \wedge (\mathcal{G}, t) \xleftarrow{Q} (\mathcal{G}, t'') \implies P = Q \wedge t' = t''$

Proof: Trivial by definition, since t' (and thus t'') is unique because functions *first-after*, *time*, *pos*, and *stepNodes* are deterministic functions; and hence $(Forward)$ and $(Backward)$ are also deterministic. ∎

Moreover, at any state s_1 where the execution of a forward step produces a new state s_2, then the execution of a backward step at s_2 always produces the original state s_1. Formally,

Theorem 3.10 (Symmetric reversibility): Given a running R-track $(\mathcal{G}, t)$,

- $(\mathcal{G}, t) \xrightarrow{P} (\mathcal{G}, t') \xleftarrow{Q} (\mathcal{G}, t'') \implies P = Q \wedge t'' = t$

Proof: Since $\textit{first-after}(\mathcal{G}, t) = \textit{last-before}(\mathcal{G}, t')$ for $n \in \textit{first-after}(\mathcal{G}, t)$ and $t' = \textit{time}(n)$ by definition, the claim holds. ∎

2) Causal-consistent reversibility of CSP: Causal-consistent reversibility [11] was first introduced in RCCS [12], a reversible variant of CCS. This form of reversibility empowers the idea that reversibility may not be necessarily symmetric in a concurrent environment provided that causality is kept consistent after every reversible step (hence, the name causal-consistent).

The main idea is that independent parallel processes should be reversed independently (thus keeping the essence of concurrence, where independent events can be executed in any order). Contrarily, when causal dependencies exist between parallel processes (e.g., synchronizations) they must be taken into account when reversing these processes. In particular, any action can be undone provided that all its consequences, if any, are undone beforehand. For instance, if two processes synchronize on event `a`, then the events that happened before `a` in any of the processes cannot be undone until all the events that happened after `a` in both processes have been undone.

Clearly, our reverse execution rule $(Backward)$ is too restrictive for causal-consistent reversibility. Even though all backward steps made by this rule are always causal-consistent, there are many possible backward schedules that are also causal-consistent but they are not allowed by the rule. For instance, in Example 3.8, the last two `b` events could be causal-consistently reversed in any order because they belong to processes that run in parallel and they are independent after the synchronization of event `a`.

In order to define proper rules for both causal-consistent forward and backward execution, we first need to define a causal-consistent version of functions *first-after* and *last-before*. Because a causal-consistent step does not necessarily follow the execution order of the R-track, the timestamp cannot be used anymore to know in what point of the execution we are (for instance, in Figure 3 timestamp 14 could be causal-consistently replayed before timestamp 12, thus changing the original order of events). Therefore, instead of timestamps we use a set with the already executed nodes N^{ex} of an R-track $\mathcal{G} = (N, E_c, E_s)$ (i.e., $N^{ex} \subseteq N$). Given two sets of executed nodes, $N^{ex1} \subset N^{ex2}$, then N^{ex2} describes a point in the execution history that is later than that of N^{ex1} (i.e., the largest timestamp t associated with any node in N^{ex1} will be strictly less than that t' associated with N^{ex2}. However, it could be possible that after N^{ex1} we causal-consistently replay a node that is not in N^{ex2} (e.g., because its execution is completely independent of the nodes in $N^{ex2} \backslash N^{ex1}$). In general, if $N^{ex1} \not\subset N^{ex2}$ and $N^{ex2} \not\subset N^{ex1}$, then there must be a node $n_1 \in N^{ex1}$, $n_1 \notin N^{ex2}$ that can be causal-consistently undone in the next step; and vice versa, there must be a node $n_2 \in N^{ex2}$, $n_2 \notin N^{ex1}$, that can be causal-consistently undone in the next step. This is ensured by the fact that (1) $n_1 \notin N^{ex1} \cap N^{ex2}$, thus n_1 belongs to a branch not executed in N^{ex2} (thus, n_1 is not causal with respect to any node in $N^{ex1} \cap N^{ex2}$, otherwise, n_1 would belong to N^{ex2}); and (2) n_1 is not causal with respect to any node in N^{ex2} (otherwise, n_1 would belong to N^{ex2}). These properties show that we can use N^{ex} to complement t (t is implicit to N^{ex}). In fact, N^{ex} contains enough information to know how developed are all branches in the R-track because an R-track

implicitly induces a Hasse diagram.

Roughly speaking, a node n' can be causal-consistently replayed after a node n if n' happened after n and all causal nodes with respect to n' have been already executed. Formally,

$$\textit{first-after}_{cc}(\mathcal{G}, N^{ex}) = \{n \in N \mid n \in \Sigma \wedge n \notin N^{ex} \wedge \\ \forall\, n' \in N \cap \Sigma, time(n') \neq time(n), n' \rightsquigarrow n \;.\; n' \in N^{ex}\}$$

The definition for $last_{cc}(\mathcal{G}, t)$ is analogous:

$$\textit{last-before}_{cc}(\mathcal{G}, N^{ex}) = \{n \in N^{ex} \mid n \in \Sigma \wedge \\ \nexists\, n' \in N^{ex} \cap \Sigma, time(n') \neq time(n) \;.\; n \rightsquigarrow n'\}$$

We can also define a causal-consistent version of function *stepNodes*:

$$\textit{stepNodes}_{cc}(\mathcal{G}, N^{ex}, n) = \textit{causalNodes}(\mathcal{G}, n) \backslash \{n' \in N \mid \\ (n' \mapsto n'') \in E_c^* \wedge n'' \in \Sigma \wedge n'' \in N^{ex}\}$$

We are now in a position to define a causal-consistent semantics:

$(Forward_{cc})$

$$\frac{n \in \textit{first-after}_{cc}(\mathcal{G}, N^{ex}) \quad P = \{pos(n') \mid n' \in \textit{stepNodes}_{cc}(n)\}}{(\mathcal{G}, N^{ex}) \xrightarrow{P} (\mathcal{G}, N^{ex} \cup \{n\} \cup \{n_s \mid (n \leftrightarrow n_s) \in E_s\})}$$

$(Backward_{cc})$

$$\frac{n \in \textit{last-before}_{cc}(\mathcal{G}, N^{ex}) \quad P = \{pos(n') \mid n' \in \textit{stepNodes}_{cc}(n)\}}{(\mathcal{G}, N^{ex}) \xleftarrow{P} (\mathcal{G}, N^{ex} \setminus (\{n\} \cup \{n_s \mid (n \leftrightarrow n_s) \in E_s\}))}$$

Example 3.11: Consider the following CSP program:

```
channel a,b
MAIN = P(a) ||{b} Q(b) ; c -> SKIP
P(x) = (x → b → a → SKIP)△x ≠ c▷(b → SKIP)
Q(x) = c → x → c → SKIP
```

and the R-track produced by the trace $\langle$cabcac$\rangle$, which can be seen in Figure 5.

For the sake of clarity, the R-track does not contain program positions, and it only shows the timestamp of the event nodes (those in bold). Every area with a set of nodes represent a graph rewriting step performed by the semantics, which corresponds to the occurrence of the event in that area. All the nodes inside an area correspond to the part of the program that induce this step (thus they are actions that can be done or undone). The steps have been numbered from 1 to 7. Steps 1 and 2, and steps 4 and 5, are mutually causal-consistent, and they could be executed in any order. For instance, if we change steps 4 and 5, then the trace would be $\langle$cabacc$\rangle$, which is a feasible trace. Similarly, if we change steps 1 and 2, then the trace would be $\langle$acbcac$\rangle$, which is another feasible trace. Note, however, that in this case, the semantics would change the nodes that belong to the areas (i.e, the *stepNodes*): the first two nodes would become *stepNodes* of event a (node 7) and would not be *stepNodes* of event c (node 4). Examples of functions *first-after*, *last-before*, *causalNodes*, and *stepNodes* are also shown.

We end with a theorem that proves the symmetric property of the forward and backward transitions. It is often known as the *loop* lemma [11].

Theorem 3.12 (Loop): For any forward transition $R \xrightarrow{P} S$, there exists a backward transition $S \xleftarrow{P} R$ and conversely.

Proof: Assume that $R = (\mathcal{G}, N_1^{ex}) \xrightarrow{P} (\mathcal{G}, N_2^{ex}) = S$. Then, by definition, there exists a node $n \in \textit{first-after}_{cc}(\mathcal{G}, N_1^{ex})$ such that

- $P = \{pos(n') \mid n' \in \textit{stepNodes}_{cc}(n)\}$, and
- $N_2^{ex} = N_1^{ex} \cup \{n\} \cup SYNC_n$.

Recall that $SYNC_n = \{n_s \mid (n \leftrightarrow n_s) \in E_s\}$. Since $n \in \textit{first-after}_{cc}(\mathcal{G}, N_1^{ex})$, we have that

1) $n \in N \cap \Sigma$,
2) $n \notin N_1^{ex}$, and
3) $\forall n' \in N, n' \in \Sigma, time(n') \neq time(n), n' \rightsquigarrow n \;.\; n' \in N_1^{ex}$.

For $S \xleftarrow{P} R$, it suffices to show that 4) $n \in \textit{last-before}_{cc}(\mathcal{G}, N_2^{ex})$, and 5) $N_1^{ex} = N_2^{ex} \setminus (\{n\} \cup SYNC_n)$.

- We first prove that 4) $n \in \textit{last-before}_{cc}(\mathcal{G}, N_2^{ex})$. To this end, we show that $n \in N_2^{ex}$ and $\nexists n' \in N_2^{ex}, n' \in \Sigma, time(n') \neq time(n) \;.\; n \rightsquigarrow n'$. It follows from the definition of *first-after*$_{cc}$ that $\textit{first-after}_{cc}(\mathcal{G}, N_1^{ex}) \subseteq N_2^{ex}$, and hence, $n \in N_2^{ex}$ by the construction of N_2^{ex}. Assume that there exists some $n' \in N_2^{ex}$ such that $n' \in \Sigma$, $time(n') \neq time(n)$, and $n \rightsquigarrow n'$. Since $n' \in N_2^{ex} = N_1^{ex} \cup \{n\} \cup SYNC_n$, we make a case analysis depending on where n' belongs to.
  - Case that $n' = \{n\} \cup SYNC_n$. In this case, we have that $time(n') = time(n)$, which is a contradiction with $time(n') \neq time(n)$.
  - Case that $n' \in N_1^{ex}$. In this case, n could not be the first (the first would be n') which is a contradiction with $n \in \textit{first-after}(N_1^{ex})$.

  Hence, $n' \notin N_2^{ex}$. This contradicts that $n' \in N_2^{ex}$.
- Next, we prove that 5) $N_1^{ex} = N_2^{ex} \setminus (\{n\} \cup SYNC_n)$. By the definition of N_2^{ex}, we have that $N_2^{ex} \setminus (\{n\} \cup SYNC_n) = N_1^{ex} \setminus (\{n\} \cup SYNC_n)$. It follows from 2) that $N_1^{ex} \setminus (\{n\} \cup SYNC_n) = N_1^{ex} \setminus SYNC_n$. Thus, it suffices to show that

  $$N_1^{ex} = N_1^{ex} \setminus SYNC_n.$$

  It is trival that $N_1^{ex} \supseteq N_1^{ex} \setminus SYNC_n$. For $N_1^{ex} \subseteq N_1^{ex} \setminus SYNC_n$, we show that $N_1^{ex} \cap SYNC_n = \emptyset$. Assume that there exists some $n_s \in N_1^{ex}$ such that $(n \leftrightarrow n_s) \in E_s$. Since $N_1^{ex} \subseteq N_2^{ex}$, we have that $n_s \in N_2^{ex}$. Then, we have that $(n, n_s) \in E^*$, and hence $n \rightsquigarrow n_s$. Since $n \notin N_1^{ex}$, we have that $n_s \neq n$, and hence $time(n_s) \neq time(n)$. It follows 4) that

  $$\nexists\, n' \in N_2^{ex} \cap \Sigma, time(n') \neq time(n) \;.\; n \rightsquigarrow n'$$

  and hence, $n \not\rightsquigarrow n_s$. This contradicts the fact that $n \rightsquigarrow n_s$.

Therefore, by definition, we have that $S \xleftarrow{P} R$.

The converse can be proved analogously. ■

IV. Implementation and empirical evaluation

The formalization presented in this paper is an event-based calculus where every step of the defined semantics goes

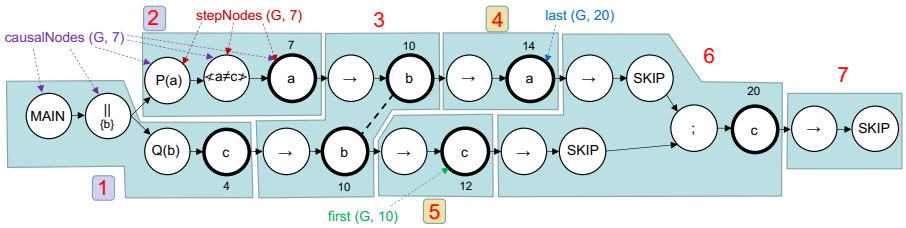

Fig. 5. An R-track from the CSP program described in Example 3.11 produced by the trace ⟨cabcac⟩.

forward or backward in the R-track until the next external event. This means that a single step of that semantics can result in an arbitrary number of rewriting steps (those that fire internal events) in the standard operational semantics [2].

Our implementation, however, strictly follows the standard operational semantics, and thus, it can perform (forward or backward) standard steps (e.g., internal choice) where external events are not necessarily involved. Therefore, the implementation is more general than the theoretical setting, and it goes into the details of handling internal events, and all rules of the standard semantics. However, this does not mean that the implementation must necessarily perform standard steps because the formalization presented has been also implemented, and thus the user can decide to perform (forward or backward) steps based on external events (as in the theoretical setting), any events (external and internal) or any single step of the standard semantics. Hence, the granularity level of the execution can be parametrised.

The implementation is a tool called *ReverCSP* that is open and publicly available (including the source code) at: https://github.com/tamarit/reverCSP. It has two main phases:

R-tracks generation. R-track generation can be done with a random execution or with a user-directed execution. Thus, the user can choose at any step the applied rule. The R-track is generated dynamically. This means that the user can perform, say, 100 random steps, then go backward, say 10 steps, and then go forward again selecting a different rule to be applied. Thus, a new R-track is dynamically generated for each new forward step performed.

R-tracks exploration. The R-track can be traversed forward or backward with either the deterministic or the causal-consistent semantics (see Figure 6). At any step, we can see the current expression, the trace, and the R-track (they can be printed as PDF).

The main bulk of the execution (track generation) is performed by csp_tracker, a submodule that executes CSP by creating a coordinating process that handles the track and one process for the first process, typically `MAIN`. The latter derives the body of the process until it ends or is deadlocked. Whenever a parallel operator is found during the execution, two processes are created, and the now parent process is left to continue its execution if a sequential composition requires it. Thus, the number of active processes is roughly equal to the number of CSP processes. The scheduling is left to the Erlang VM, and is not tuned by csp_tracker in any specific direction.

```
Current expression:
(P [|{|a|}|] R)

These are the available options:
1 .- P
2 .- R
3 .- Random choice.
4 .- Random forward-reverse choice.
5 .- See current trace.
6 .- Print current R-track.
7 .- Reverse evaluation.
8 .- Undo.
9 .- Roll back.
0 .- Finish evaluation.
What do you want to do?
[1/2/3/4/5/6/7/8/9/0]:
```

Fig. 6. Main menu of *ReverCSP*

When a backwards step is taken, reverCSP traverses the R-track to reach the appropriate state and display it to the user. When a forwards step is taken that is present in the graph, reverCSP traverses the graph instead of re-running the simulation.

The rest of this section provides technical details about the implementation so that researchers or developers that want to implement this technique (e.g., for other language) can see the rationale and algorithms used. We use a functional language to describe the implementation.

Given a computation, i.e., an R-track $\mathcal{G} = (N, E_c, E_s)$, the set of reversible actions can be obtained with the following function call: $reversible(Leaves, \emptyset, \emptyset)$ where $Leaves = \{n \mid n \in N, \nexists\ n' \in N\ .\ (n \mapsto n') \in E_c\}$, which is defined in Figure 7, Equation 1.

All parameters of function $reversible$ are subsets of N: L is the set to be analysed, R is an accumulator parameter with the reversible actions (the final output), and C is a list of candidates that should wait for the rest of nodes that are synchronised with them. Function irp (is relevant point) should return $true$ or $false$ whether the node is relevant to stop or not. For instance, for an event node it should return $true$, while for a node of a parallelism operator, it should return $false$. Function $reversible$ returns a set of sets, where each set represents an undoable action and all the nodes involved in that action.

The result of function $reversible$ can be used to undo one of the undoable actions. The function that removes an action Sel from an R-track $\mathcal{G}$ is defined as follows:

$$reversible(L,R,C) = \begin{cases} R & \text{if } L = \emptyset \\ reversible(L\backslash\{n\}, R \cup \{Sync \cup \{n\}\}, C\backslash Sync) & \text{if } \exists n \in L \wedge irp(n) \wedge Sync \cap C = Sync \\ reversible(L\backslash\{n\}, R, C \cup \{n\}) & \text{if } \exists n \in L \wedge irp(n) \wedge Sync \cap C \neq Sync \\ reversible((L \cup Prev)\backslash\{n\}, R, C) & \text{if } \exists n \in L \ \wedge \ not(irp(n)) \end{cases} \quad (1)$$

$$\text{where } Sync = \{n' \mid n' \in N, (n \leftrightarrow n') \in E_s\}, Prev = \{n' \mid n' \in N, (n' \mapsto n) \in E_c\}$$

$$sft(s,T,t) = \begin{cases} s & \text{if } \forall n \in N \ . \ t > time(n) \\ sft(s',T,t+1) & \text{if } \exists n \in N \ . \ time(n) \leq t \wedge \exists s \rightarrow s' = (\_, T_c, \_, \_) \in der(s) \ . \ tt(T,t) = T_c \\ sft(s,T,t+1) & \text{if } \exists n \in N \ . \ time(n) \leq t \wedge \nexists s \rightarrow s' = (\_, T_c, \_, \_) \in der(s) \ . \ tt(T,t) = T_c \end{cases} \quad (2)$$

$$tdrs(n, T = (N, E_c, E_s)) = \begin{cases} (pos, \bullet) & \text{if } n = \bullet_{pos} \\ (next(n), n) & \text{if } n \notin \{||, |||\} \wedge \nexists n' \in N.(n \mapsto n') \in E_c \\ tdrs(n', T) & \text{if } n \notin \{||, |||\} \wedge \exists n' \in N.((n \mapsto n') \in E_c \\ & \quad \wedge \nexists n'' \in N.(n \mapsto n'') \in E_c \wedge n'' \neq n') \\ (n'_l \ pos(n)_{(pos(n),p'_l,p'_r)} \ n'_r, n) & \text{if } n \in \{||, |||\} \end{cases} \quad (3)$$

where $(n_l, n_r) = lr(n,T) \wedge (n'_l, p_l) = tdrs(n_l, T) \wedge (n'_r, p_r) = tdrs(n_r, T) \wedge ((p'_l = n \wedge p_l = \bullet) \vee (p'_l = p_l \wedge p_l \neq \bullet)) \wedge ((p'_r = n \wedge p_r = \bullet) \vee (p'_r = p_r \wedge p_r \neq \bullet))$, and function $next(n)$ returns the expression that follows the expression of node n:

$$next(n) = \begin{cases} pos(n).2 & \text{if } n = \ \rightarrow \\ n.\Lambda & \text{if } n \in ProcessNames \\ \top & \text{if } n = SKIP \\ \bot & \text{if } n = STOP \\ pos(n).b & \text{if } n = \sqcap.b \end{cases} \quad (4)$$

$$lr(n, T = (N, E_c, E_s)) = \begin{cases} (n_l, n_r) & \text{if } \exists n_l, n_r \in N \ . \ (n \mapsto n_l) \in E_c \wedge (n \mapsto n_r) \in E_c \\ & \quad \wedge pos(n_l) = pos(n).1 \wedge pos(n_r) = pos(n).2 \\ (n_l, \bullet_{pos(n).2}) & \text{if } \exists n_l \in N \ . \ ((n \mapsto n_l) \in E_c \wedge pos(n_l) = pos(n).1) \\ & \quad \wedge \nexists n_r \in N \ . \ ((n \mapsto n_r) \in E_c \wedge pos(n_r) = pos(n).2) \\ (\bullet_{pos(n).1}, n_r) & \text{if } \nexists n_l \in N \ . \ ((n \mapsto n_l) \in E_c \wedge pos(n_l) = pos(n).1) \\ & \quad \wedge \exists n_r \in N \ . \ ((n \mapsto n_r) \in E_c \wedge pos(n_r) = pos(n).2) \\ (\bullet_{pos(n).1}, \bullet_{pos(n).2}) & \text{if } \nexists n_l \in N \ . \ ((n \mapsto n_l) \in E_c \wedge pos(n_l) = pos(n).1) \\ & \quad \wedge \nexists n_r \in N \ . \ ((n \mapsto n_r) \in E_c \wedge pos(n_r) = pos(n).2) \end{cases} \quad (5)$$

Fig. 7. Definition of multiple functions used throughout the implementation.

$$remove(\mathcal{G} = (N, E_c, E_s), Sel) = (N', E'_c, E'_s)$$

$$\begin{aligned} \text{where } N' &= N\backslash(\bigcup_{n \in Sel} \{n\} \cup \{n' \mid n' \in N, (n \mapsto n') \in E_c^*\} \cup \\ &\quad \{n' \mid n' \in N, (n \leftrightarrow n') \in E_s^*\}) \\ E'_c &= E_c\backslash\{(n \mapsto n') \mid (n \mapsto n') \in E_c, n \notin N' \vee n' \notin N'\} \\ &\quad \text{in the following denoted as: } E'_c = E_c|_{N'} \\ E'_s &= E_s\backslash\{(n \leftrightarrow n') \mid (n \leftrightarrow n') \in E_s, n \notin N' \vee n' \notin N'\} \\ &\quad \text{in the following denoted as: } E'_s = E_s|_{N'} \end{aligned}$$

When an action is undone with the *remove* function we have a new R-track from which it must be possible to perform a new action (i.e., step forward). Therefore, we need a mechanism to regenerate the state of the tracking semantics given this new R-track. In this way, we can continue the computation from the concrete state given by this new R-track. This has been implemented using function *sft* (state from R-track). This function uses function $der(s)$, which returns all possible derivations of state s using the tracking semantics. Additionally, function tt (R-track in time) is used to calculate the R-track $\mathcal{G}'$ in a given time t using the R-track $\mathcal{G}$ of the whole computation. Its definition is the following:

$$tt(\mathcal{G} = (N, E_c, E_s), t) = (N', E'_c, E'_s) = \mathcal{G}'$$

$$\begin{aligned} \text{where } N' &= \{n \mid n \in N \ . \ time(n) \leq t\} \\ E'_c &= E_c|_{N'} \\ E'_s &= E_s|_{N'} \end{aligned}$$

We can now define function *sft*, which allows us to start the computation in the same point where the computation associated with the R-track stopped, as can be seen in Figure 7, Equations 2, 3 and 4.

Finally, function *lr* (Figure 7, Equation 5) returns the left and right nodes of a bifurcation operator. In case any (or both) of the branches was not unfolded, the symbol • is used to denote this fact. The • is labelled with the first position of the non-evaluated branch.

A. Empirical evaluation

In order to precisely quantify the performance of ReverCSP, we conducted several experiments to evaluate the size and the time needed to produce tracks. For the evaluation, a set of heterogeneous benchmarks was selected from public CSP repositories. All of them have been previously used to evaluate other CSP techniques and tools. The selected benchmarks ensure that they cover all the CSP 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. Each benchmark includes a header describing it

and providing details about the authors and a reference to its source.

The empirical evaluation strictly followed the method and guidelines proposed in [13], [14]. The same hardware configuration was used to assess all benchmarks: Intel® Xeon® E5504 Processor (2 Ghz, 4MB Cache, 4 cores) with 16GB RAM. In order to avoid interference of other programs, all processes except ReverCSP were stopped while the benchmarks were running. Every benchmark was run 1001 times. In all cases, the first iteration was discarded to avoid the influence of cached data persisting in the disk, or dynamically loaded libraries stored in physical memory. Thus, we finally obtained 1000 statistical values for each benchmark. This process was repeated for the 10 benchmarks. In order to study the effect of statistical dispersion, we computed both the harmonic and the arithmetic mean. The former was sufficiently low so that we could use the arithmetic mean in our table results.

Because the benchmarks could produce infinite executions (e.g. due to livelock processes) we automatically stopped them after a timeout of 2 seconds. Thanks to the good scalability of the tool, this threshold was enough to produce long tracks with many parallel processes and more than 1500 nodes. This threshold allowed us to compare the track produced by different CSP specifications (with different number of synchronizations and parallel processes, and different levels of complexity, etc.) executed exactly the same time. The results are shown in Table I.

In the table, we use symmetric 0.99 confidence intervals. The meaning of the columns is the following: *Benchmark* is the name of the benchmark. *Runtime* is the time that the benchmark was executed before it eventually ended or before it was stopped. *#Nodes* is the number of nodes that compose the track. *#Edges* is the number of (control and synchronization) edges that compose the track.

B. Study of the time and memory overhead

The generation of R-tracks introduces a measurable overhead in both time and memory, as opposed to simulating CSP without storing any state information apart from the necessary to run the following step. To determine the scale of this overhead, we conducted another empirical evaluation, measuring the time and memory used.

The benchmark suite used is very similar to the one used in the performance evaluation, except for a small change in *Buses* and the removal of *ATM* and *ABP*, and can be found alongside with the code and appropriate compilation flags to reproduce the evaluation in the git tags `TDPS_bench_time` and `TDPS_bench_memory` in our repository (https://github.com/mistupv/csp_tracker). The same hardware configuration was used to assess all benchmarks: Intel® Xeon® E3-1220 v3 @ 3.10GHz Processor[2] with 8GB DDR3 RAM at 1333MHz (single channel). In order to avoid interference from other programs, all other non-essential processes were stopped.

Two modes of execution are present in the code: `run` and `track`. The latter is the one used to generate R-tracks, which can be then interactively traversed by the user. The former is very similar, but skips all possible track-generating operations, and simulates a CSP interpreter that executes a CSP specification as fast as possible.

Each time benchmark was run 1001 times in each mode: `run` and `track`. The same Erlang VM was used for each block of 1001 benchmarks, but the first iteration was discarded to avoid the delay introduced by loading the required code and libraries from disk to RAM.

Each memory benchmark was run 1000 times in each of the aforementioned modes. In this case, each iteration was run in its own Erlang VM, as to guarantee that no remnants of the previous iteration lived on in memory. Before each iteration, the necessary code and libraries were manually loaded as to obtain the most precise measurement possible.

Thus, we obtained 1000 statistical values per benchmark and mode for each metric (time and memory consumption). A timeout of one second was established, as some processes are infinitely long, and would not stop otherwise. As a way to compare both modes of execution, we measured the number of operations that each process took, so that the `track` and `run` versions could be easily compared. Our results are shown in Table II.

In the table, we show the 0.99 error margins as a percentage. The first column is the name of the file; the second and third show the consumption of *kilobytes per operation* taken, for both `run` and `track` modes. *Memory overhead* displays the ratio, showing that generating R-tracks is several times more memory-intensive than running the CSP specification. *Microseconds per operation* shows the average time needed per operation for each mode, and *time overhead* displays the ratio, with meaning similar to *memory overhead*.

As can be seen, there is an overhead of up to an order of magnitude in general, but three of the benchmarks have overheads of two orders of magnitude for time and three for memory. These three (`Buses`, `Loop` and `ProdCons`) are not only infinite but have an ever-increasing number of active processes, which increases the size of the R-track considerably. The other benchmarks keep the number of active processes stable, even if they do run indefinitely.

V. Related Work

Reversibility [7] is the ability of a program to be executed both forward and backward, thus having the possibility to go back to past states. This ability has been studied in most languages (e.g., CSS [15], π-calculus [16], Erlang [17], and CSP [18], among many others) and has different applications such as program comprehension and debugging [19], quantum computing [20], biological systems modelling [21], etc.

In the specific case of debugging, different approaches have been defined for rollback-recovery [22], [23], whose most simple instance is the *undo* button, which restores the immediate previous state. While reversibility in a sequential system is understood as the reverse of a set of actions in the opposite order as they were done (see, e.g., [24]), reversibility in a concurrent system is not intuitive at all, because the order of actions is often undefined (e.g., many

[2]See the full specification at https://ark.intel.com/content/www/us/en/ark/products/75052/intel-xeon-processor-e3-1220-v3-8m-cache-3-10-ghz.html

TABLE I
SIZE OF THE TRACKS GENERATED WITH A GIVEN RUNTIME

Benchmark	Runtime (ms)	#Nodes	#Edges
ABP.csp	$[2208.16\ 2209.25\ 2210.34]$	$[1505.61\ 1506.17\ 1506.73]$	$[1303.10\ 1303.63\ 1304.17]$
ATM.csp	$[630.17\ 690.18\ 750.19]$	$[364.09\ 405.64\ 447.19]$	$[300.74\ 334.67\ 368.61]$
Buses.csp	$[126.40\ 127.19\ 127.97]$	$[22.00\ 22.00\ 22.00]$	$[18.00\ 18.00\ 18.00]$
CPU.csp	$[189.97\ 190.74\ 191.51]$	$[87.43\ 87.76\ 88.09]$	$[71.23\ 71.50\ 71.77]$
Disk.csp	$[209.07\ 210.10\ 211.13]$	$[148.50\ 148.74\ 148.98]$	$[123.59\ 123.78\ 123.78]$
Loop.csp	$[2133.02\ 2133.99\ 2134.96]$	$[1537.53\ 1538.34\ 1539.14]$	$[1230.05\ 1230.69\ 1231.35]$
Oven.csp	$[238.64\ 241.92\ 245.20]$	$[157.16\ 163.37\ 169.59]$	$[162.68\ 169.33\ 175.98]$
ProdCons.csp	$[2134.59\ 2135.43\ 2136.27]$	$[1535.43\ 1536.09\ 1536.75]$	$[1228.08\ 1228.61\ 1229.15]$
ReadWrite.csp	$[2148.76\ 2149.71\ 2150.65]$	$[1475.85\ 1476.56\ 1477.28]$	$[1252.47\ 1253.34\ 1254.22]$
Traffic.csp	$[165.34\ 166.35\ 167.36]$	$[61.18\ 64.37\ 67.56]$	$[47.73\ 50.13\ 52.53]$
Average	$[1018.41\ 1025.49\ 1019.76]$	$[689.478\ 694.90\ 700.33]$	$[573.77\ 578.37\ 582.98]$

TABLE II
MEMORY AND TIME CONSUMPTION OVERHEAD INTRODUCED BY R-TRACKS

Benchmark	kilobytes per operation		Memory	microseconds per operation		Time
	`run`	`track`	overhead	`run`	`track`	overhead
Buses.csp	$0.26 \pm 0.80\%$	$36.28 \pm 1.19\%$	140.375	$37.62 \pm 0.22\%$	$2312.07 \pm 0.04\%$	61.466
CPU.csp	$11.38 \pm 0.63\%$	$38.51 \pm 1.23\%$	3.384	$22.86 \pm 2.10\%$	$117.63 \pm 1.18\%$	5.146
Disk.csp	$8.83 \pm 0.52\%$	$45.81 \pm 1.85\%$	5.188	$24.63 \pm 1.50\%$	$171.01 \pm 0.79\%$	6.944
Loop.csp	$0.04 \pm 1.33\%$	$33.83 \pm 0.85\%$	930.505	$12.50 \pm 0.19\%$	$1314.21 \pm 0.06\%$	105.171
Oven.csp	$13.77 \pm 0.77\%$	$76.86 \pm 1.88\%$	5.580	$30.66 \pm 1.84\%$	$344.66 \pm 4.43\%$	11.242
ProdCons.csp	$0.04 \pm 1.87\%$	$33.24 \pm 0.86\%$	890.739	$15.03 \pm 0.04\%$	$1326.11 \pm 0.09\%$	88.215
ReadWrite.csp	$13.78 \pm 1.04\%$	$69.94 \pm 0.97\%$	5.077	$707.12 \pm 0.32\%$	$2492.11 \pm 0.31\%$	3.524
Traffic.csp	$37.92 \pm 1.46\%$	$96.18 \pm 1.57\%$	2.536	$53.12 \pm 2.72\%$	$163.20 \pm 2.56\%$	3.072

actions can happen concurrently). For this reason, reversibility is particularly useful to debug concurrent languages, because there is no guarantee that a bug that appears in the original computation is replayed inside the debugger. This problem is usually tackled by so-called replay debugging [25]. Our replay debugger uses a new data structure called *track* that allows us to reconstruct all states in the execution in both directions. Tracks were introduced in [10] as a means to record executions. The original idea of that paper is that they could be used for program comprehension and tracing. The idea of using them for reversibility was proposed in [18]. In this paper we extend the work done in [18] with several new ideas and a formal theory about how to use R-tracks for reversibility. This theory includes a formal reversibility semantics for CSP. With that purpose, we have slightly extended the original notion of tracks so that they can be used to replay computations forward and backward. This new track is called R-track. The reversible semantics proposed is completely novel with respect to [10], The new implementation is conservative with respect to the original tracks because it can generate standard CSP tracks, which are useful for, e.g., program comprehension; but it also generates the extended tracks proposed here. With this extension our debugger can replay the execution in any causal-consistent order.

An R-track defines a partial order through the causality relation $n_1 \rightsquigarrow n_2$, which is clearly reflexive, antisymmetric and transitive. Therefore, the model proposed based on R-tracks is a model for true concurrency based on a non-interleaving semantics which explicitly represents causality in the R-track. Other semantics for reversibility that are related to our work are the non-interleaving semantics for CSS [15] and π-calculus [26].

A system that is similar to ours, but for Erlang, and using a different notion of tracks, is CauDEr, a causal-consistent reversible debugger for Erlang [27]. The functionality of CauDEr is similar to our tool because it allows undoing actions in a causal-consistent manner. Causal-consistent reversibility was introduced in [12], with RCCS, a reversible calculus for CCS. It introduced a mechanism to attach memories to threads to keep history information. Later interesting approaches are [19], [23]. A survey that very nicely explains the evolution of this field can be found in [11].

Other approaches that are related to our work are [28], [29] and [25]. First, in [28], the authors introduced a modular framework for defining causal-consistent reversible extensions of concurrent models and languages. This work has inspired some of our ideas to define the extended tracks. The work by Brown and Sabry [29] is another interesting work that also proposes two semantic reversible models for a CSP-based language embedded in Scala. They also report that their implementation was evaluated and showed that a practical reversible distributed language can be efficiently implemented in a fully distributed manner. Unfortunately, the implementation is not publicly available. Finally, the work by Lanese et al. [25] proposes a novel approach for replay debugging that is called *controlled causal-consistent replay*. It is called "controlled" in the sense that the debugger shows all and only the causes of an error. This approach is also related to causally-consistent dynamic slicing [30]. However, while our approach uses tracks to traverse the computation forward and backward, dynamic slicing uses a trace to extract the part of the code (the so-called slice) that could influence a given behaviour. Another

difference is the target language: pi calculus vs CSP.

VI. Conclusions

This work introduces R-tracks, an extension of standard CSP tracks so that they can be used to replay and to reverse computations. Given an R-track, we propose two calculus for reversibility: (1) the first one can reverse computations so that events are executed exactly in the opposite temporal order in which they happened in the original computation; and (2) the second calculus can reverse computations according to the causal-consistency principle. This second calculus is particularly useful for debugging because one can do or undo an event and analyze what other events must be executed before (or after) it (because there exists a causal relation). In both calculus, we have proven important properties such as deterministic reversibility and symmetric reversibility.

The extension of the tracks and the calculus proposed have been implemented in a publicly available debugger for CSP. This tool has interesting extensions. For instance, it allows the users to dynamically interweave the reversal of computations and the generation of tracks. In particular, we have two phases:

1) Execution of the program and generation of the associated R-track (as a side effect)

2) Exploration of the R-track (both forward and backward)

Our implementation goes beyond the traditional postmortem analysis: (1) $\rightarrow$ (2). It allows to do a dynamic (reversible) execution of CSP from any state: (1) $\rightarrow$ (2) $\rightarrow$ (1) $\rightarrow$ (2) ... For this, the tool implements a mechanism to obtain an initial state from a given R-track/timestamp. This state is animated in the extended semantics of CSP to generate R-tracks, and thus, the normal execution can continue (generating a new R-track from the given state).

Another interesting feature of the debugger is that it implements *step-by-step* reversibility. Step-by-step means that each single step can be undone, as opposed, e.g., to checkpointing where many steps are undone at once. But, it can also work as a checkpointing debugger. It is enough to introduce a new event called `checkpoint` in the specification, and rollback all the computation steps until the timestamp when `checkpoint` happened.

Acknowledgement

A preliminary version of this work was presented at the 12th Conference on Reversible Computation [31]. The authors want to thank the anonymous reviewers for their useful comments and constructive feedback that helped us to improve this work.

References

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

[2] A. W. Roscoe, *The Theory and Practice of Concurrency*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1997.

[3] P. Ladkin and B. Simons, "Static deadlock analysis for csp-type communications," in *In: Fussell D.S., Malek M. (eds) Responsive Computer Systems: Steps Toward Fault-Tolerant Real-Time Systems. The Springer International Series in Engineering and Computer Science, vol 297. Springer, Boston, MA*, 1995.

[4] H. Zhao, H. Zhu, F. Yucheng, and L. Xiao, "Modeling and verifying storm using csp," in *Proceedings of the 19th International Symposium on High Assurance Systems Engineering (HASE)*, ser. HASE '19. Hangzhou, China: IEEE Computer Society, 2019. [Online]. Available: 10.1109/HASE.2019.00037

[5] M. Conserva Filhoa, M. Oliveira, A. Sampaio, and A. Cavalcanti, "Compositional and local livelock analysis for csp," *Information Processing Letters*, vol. 133, pp. 21–25, May 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0020019018300036

[6] Y. Fang, H. Zhu, F. Zeyda, and Y. Fei, "Modeling and analysis of the disruptor framework in csp," in *Proceedings of the 8th Annual Computing and Communication Workshop and Conference (CCWC)*, ser. CCWC '18. Las Vegas, NV, USA: IEEE Computer Society, 2018. [Online]. Available: 10.1109/CCWC.2018.8301703

[7] B. Aman et al., "Foundations of reversible computation," in *Ulidowski I., Lanese I., Schultz U., Ferreira C. (eds) Reversible Computation: Extending Horizons of Computing. RC 2020*, ser. Lecture Notes in Computer Science, vol. 12070. Springer, 2020.

[8] M. Llorens, J. Oliver, J. Silva, and S. Tamarit, "Dynamic slicing of concurrent specification languages," *Parallel Computing*, vol. 53, pp. 1–22, 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819116000363

[9] R. Landauer, "Irreversibility and heat generation in the computing process," *IBM Journal of Research and Development*, vol. 5, pp. 183–191, 1961.

[10] M. Llorens, J. Oliver, J. Silva, and S. Tamarit, "Tracking csp computations," *Journal of Logical and Algebraic Methods in Programming*, vol. 102, pp. 138–175, 2019.

[11] I. Lanese, C. Antares Mezzina, and F. Tiezzi, "Causal-consistent reversibility," *Bulletin of the EATCS*, vol. 114, p. 17, 2014.

[12] V. Danos and J. Krivine, "Reversible communicating systems," in *Proceedings of the Fifteenth International Conference on Concurrency Theory (CONCUR'04)*, ser. LNCS, vol. 3170. Springer, 2004, pp. 292–307.

[13] A. Georges, D. Buytaert, and L. Eeckhout, "Statistically rigorous Java performance evaluation," *SIGPLAN Not.*, vol. 42, no. 10, pp. 57–76, October 2007. [Online]. Available: http://doi.acm.org/10.1145/1297105.1297033

[14] I. Souilah, A. Francalanza, and V. Sassone, "A formal model of provenance in distributed systems," *First Works. on Theory and Practice of Provenance (TAPP'09)*, pp. 1–21, 2009.

[15] P. Degano, R. De Nicola, and U. Montanari, "A partial ordering semantics for ccs," *Theoretical Computer Science*, vol. 75, no. 3, pp. 223 – 262, 1990. [Online]. Available: http://www.sciencedirect.com/science/article/pii/030439759090095Y

[16] I. Lanese, C. Mezzina, A. Schmitt, and S. J.B., "Controlling reversibility in higher-order pi," in *In: Katoen JP., König B. (eds) CONCUR 2011 – Concurrency Theory. CONCUR 2011. Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-23217-6_20*, vol. 6901. Springer, 2011.

[17] I. Lanese, N. Nishida, A. Palacios, and G. Vidal, "A theory of reversibility for erlang," 2018.

[18] C. Galindo, N. Nishida, J. Silva, and S. Tamarit, "ReverCSP: Time-travelling in CSP computations," in *Proceedings of the 12th International Conference on Reversible Computation (RC 2020)*, ser. Lecture Notes in Computer Science, I. Lanese and M. Rawski, Eds. Springer, 2020, to appear.

[19] E. Giachino, I. Lanese, and C. A. Mezzina, "Causal-consistent reversible debugging," in *Proceedings of FASE 2014*, April 2014, pp. 370–384.

[20] T. Altenkirch and J. Grattage, "A functional quantum programming language," *LICS*, pp. 249—-258, 2005. [Online]. Available: doi: 10.1109/LICS.2005.1

[21] L. Cardelli and C. Laneve, "Reversible structures," *CMSB, ACM. doi:10.1145/2037509.2037529*, vol. 321, pp. 131–140, 2011.

[22] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, "A survey of rollback- recovery protocols in message-passing systems," *ACM Comput. Surv.*, vol. 34, no. 3, pp. 375—-408, 2002.

[23] E. Giachino, I. Lanese, C. A. Mezzina, and F. Tiezzi, "Causal-consistent rollback in a tuple-based language," *Journal of Logical and Algebraic Methods in Programming*, vol. 88, pp. 99 – 120, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2352220816301109

[24] I. Phillips, I. Ulidowski, and S. Yuen, "A reversible process calculus and the modelling of the erk signalling pathway," in *Proceedings of the 4th International Conference on Reversible Computation (RC'12)*, ser. LNCS 7581. Springer, 2012, pp. 218–232.

[25] I. Lanese, A. Palacios, and G. Vidal, "Causal-consistent replay debugging for message passing programs," in *In: 39th International Conference on Formal Techniques for Distributed Objects, Components, and Systems (FORTE 2019). Copenhagen, Denmark.*, June 2019, pp. 167–184.

[26] P. Degano and C. Priami, "Non-interleaving semantics for mobile processes," *Theor. Comput. Sci.*, vol. 216, no. 1–2, p. 237–270, Mar. 1999. [Online]. Available: https://doi.org/10.1016/S0304-3975(99)80003-6

[27] I. Lanese, N. Nishida, A. Palacios, and G. Vidal, "Cauder: A causal-consistent reversible debugger for erlang," *Functional and Logic Programming*, vol. 10818, pp. 247–263, 2018.

[28] A. Bernadet and I. Lanese, "A Modular Formalization of Reversibility for Concurrent Models and Languages," in *ICE 2016*, ser. EPTCS, Heraklion, Greece, Jun. 2016. [Online]. Available: https://hal.inria.fr/hal-01337423

[29] G. Brown and A. Sabry, "Reversible communicating processes," *Electronic Proceedings in Theoretical Computer Science*, vol. 203, p. 45–59, Feb 2016. [Online]. Available: http://dx.doi.org/10.4204/EPTCS.203.4

[30] R. Perera, D. Garg, and J. Cheney, "Causally consistent dynamic slicing," in *Proceedings of the 2016 International Conference on Concurrency (CONCUR'16)*, ser. LIPIcs, J. Desharnais and R. Jagadeesan, Eds., vol. 59, chloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016, pp. 18:1—-18:15.

[31] C. Galindo, N. Nishida, S. Tamarit, and J. Silva, "Revercsp: Time-travelling in csp computations," in *Reversible Computation*, I. Lanese and M. Rawski, Eds. Cham: Springer International Publishing, 2020, pp. 239–245. [Online]. Available: https://doi.org/10.1007/978-3-030-52482-1_14

Carlos Galindo received the Bachelor's Degree (2018) in computer science from Universitat Politècnica de València (UPV), Spain; completing his Bachelor's Thesis abroad while taking part in a student exchange program in the Eidgenössische Technische Hochschule Zürich (ETHz), Switzerland. Afterwards, he completed a Master's Degree (2019) specialization in software engineering. In 2019, he joined the VRAIN research institute, and began working on his Ph.D. He is currently working on his Ph.D. in Computer Science, in the area of program slicing and static analysis.

Naoki Nishida graduated with the D.E. degree from the Graduate School of Engineering at Nagoya University in 2004. He became a Research Associate in the Graduate School of Information Science at Nagoya University in 2004. From April 2007, he has been an Assistant Professor at Nagoya University, and he was a visiting researcher in DSIC at Universitat Politècnica de València from July 2011 to December 2011. From April 2013, he has been an Associate Professor at Nagoya University. He received the Best Paper Award from IEICE in 2011. He is interested in program inversion, theorem proving, term rewriting, and program verification. He is a member of IEICE, IPSJ, and JSSST.

Salvador Tamarit finished his academic activity with a Postdoc Researcher at the Computer Science Faculty of the Universitat Politècnica de València (UPV). Dr. Tamarit earned his Ph.D. in the area of analysis and transformation of concurrency languages. After earning his Ph.D., he participated in a European Project at the Universidad Politécnica de Madrid (UPM), where he applied his knowledge in the transformation of code to automatically adapt code for multi-platform High Performance systems. In the meantime, he also worked in automatic debugging and analysis of Erlang programs. Currently, he is working in the industry.

Josep Silva received the B.S. (2001) and Ph.D. (2006) in computer science from Universitat Politècnica de València, Spain. Afterwards, he completed a M.S. (2007) specialization in software engineering. Since 2001, he has been a professor in three universities. Currently, he is an Associate Professor at UPV, and an Aggregate Professor at UNED. In 2019, he joined the VRAIN research institute as a member of the MiST research group on program analysis and transformation. Dr. Silva has published over 90 papers in international journals and conferences. His current research is centred on debugging, program slicing, and testing.