ReverCSP: Time-travelling in CSP computations

Carlos Galindo[1][0000−0002−3569−6218], Naoki Nishida[2][0000−0001−8697−4970], Josep Silva[1][0000−0001−5096−0008], and Salvador Tamarit[1][0000−0001−5103−4153]

[1] Departament de Sistemes Informàtics i Computació
Universitat Politècnica de València
Camino de Vera sn, E-46022 Valencia, Spain
[2] Graduate School of Informatics
Nagoya University
Furo-cho, Chikusa-ku, 464-8603 Nagoya, Japan.

Abstract. This paper presents *reverCSP*, a tool to animate both forward and backward CSP computations. This ability to reverse computations can be done step by step or backtracking to a given desired state of interest. *reverCSP* allows us to reverse computations exactly in the same order in which they happened, or also in a causally-consistent way. Therefore, *reverCSP* is a tool that can be especially useful to comprehend, analyze, and debug computations. *reverCSP* is an open-source project publicly available for the community. We describe the tool and its functionality, and we provide implementation details so that it can be reimplemented for other languages.

Keywords: Reversible computations · CSP · Tracing.

1 Introduction

The Communicating Sequential Processes (CSP) is nowadays one of the must used process algebras [16]. The analysis of CSP computations has traditionally been based on the so-called *CSP traces*. Roughly, CSP traces are a representation to specify all possible computations that may occur in a system, and they are represented with sequences of events. Among the different analyses defined over traces we have security analysis [6], livelock analysis [3], and deadlock analysis [7, 17].

Unfortunately, CSP traces are not very appropriate for debugging because they do not relate the computations with the source code. For this reason, a data structure called CSP track [12] was defined to overcome that problem. CSP tracks were originally conceived for program comprehension and debugging because they can represent forward CSP computations with the advantage that every single step of the operational semantics is associated with the positions in the source code (i.e., initial and final line and column) of the literals of the specification participating in that step. This means that, with a CSP track, one can see directly in the source code the parts that are being executed.

Example 1. Consider the following CSP specification:[3]

[3] Those readers non familiar with the CSP syntax are referred to [16], where all CSP syntax constructs are explained.

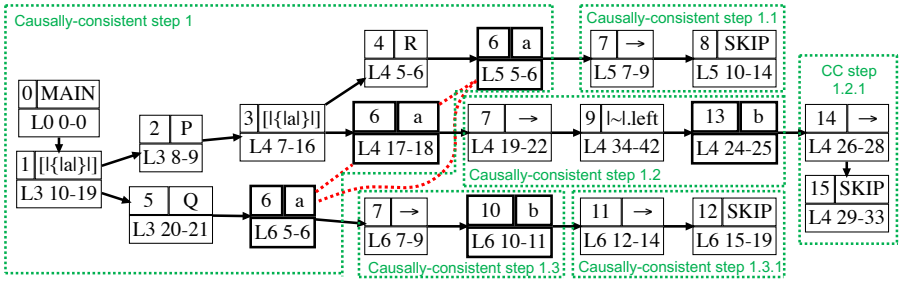

Fig. 1. Extended track of the computation produced by the trace ⟨abb⟩ in *reverCSP*.

```
channel a,b

MAIN = P ||{a} Q

P = R ||{a} a → (b → SKIP ⊓ Q)

R = a → SKIP

Q = a → b → SKIP
```

The only possible traces of this specification are: {⟨⟩,⟨a⟩,⟨ab⟩,⟨abb⟩}

If we consider the trace ⟨abb⟩, it can be produced by two different computations due to the non-deterministic evaluation order of the processes. While the first event (a) is deterministic, the b events are not (they could correspond to either process P or Q). Therefore, a trace ⟨abb⟩ does not give information about what parts of the computation have been executed and in what order.

In contrast, if we observe the track in Figure 1, we can see that it represents the source code literals inside nodes (at the top right); each node is labelled with its associated timestamp (at the top left) and they contain pairs line-column to uniquely identify the literals in the CSP specification. Synchronizations are represented with a dashed edge. For the time being the reader can ignore the green text and lines.

In this paper we present a new tool called *reverCSP* that uses an extension of CSP tracks to animate and reverse computations. We explain how to download and install the tool, and we explain its functionality and architecture.

2 Recording the history of a CSP computation

According to the Landauer's embedding principle [8] a record of a computation can make that computation reversible. In order to record CSP computations we have defined an extension of CSP tracks [12] so that they also store the exact

time when each literal in the track was executed. This gives us the ability to know exactly in what order where the literals executed and, thus, to reverse computations. Observe in Figure 1 that each node has a label with a timestamp that represents the instant where this node was generated. Therefore, synchronized events have the same timestamp.

With the timestamp we can serialize the program. For instance, if we only focus on event nodes (those in bold) then it is trivial to generate the associated trace ⟨`abb`⟩ following the sequence: `(6,a)`→`(10,b)`→`(13,b)`. Timestamps together with synchronizations also allow us to define a causally-consistent relation between nodes. This relation allows us to perform (forward and backward) causally-consistent steps. These steps group a set of nodes that must happen before a given action (a visible event or the end of the computation represented with `SKIP` or `STOP`) and after another action that already happened.

Example 2. Consider again the track in Figure 1. Those nodes that belong to the same causally-consistent step have been grouped inside an area marked with a dotted green line. The causal relation is represented by the identifier of the causally-consistent steps. Step X.Y cannot be undone until any suffix of X.Y has been undone. This means that steps 1.1, 1.2, and 1.3 must be undone (in any order) before undoing step 1. Similarly, step 1.2.1 must be undone before undoing step 1.2. Steps 1.2.1 and 1.3.1 can be undone in any order. All this information is automatically computed by *reverCSP* and used to control that steps are (un)done (and offered to the user) in the correct order.

3 The system *reverCSP*

3.1 Downloading and installation

The *reverCSP* system is open-source and free. It can be downloaded from: `https://github.com/tamarit/reverCSP`. The system can be run either on Linux or in a Docker container. The later is the simplest, as the user only needs to install docker and run the following commands:

```
$ git clone --recursive https://github.com/tamarit/reverCSP
$ docker build -t reverCSP .
$ docker run -it -v $PWD/examples:/reverCSP/examples \
    -v $PWD/output:/reverCSP/output --rm reverCSP
```

Then, from within the shell inside the docker container, the user can run the script `reverCSP`, accompanied by the path to a CSP specification file, as can be seen in Figure 2. The two volumes exposed to docker (the `-v` option) allow the user to view the generated PDF files in the output folder and to add new specifications to be analyzed.

The system uses the Erlang/OTP framework[4] to animate CSP specifications, and it (optionally) uses Graphviz[5] to produce PDF outputs of the tracks. Oth-

[4] `https://www.erlang.org`
[5] `https://www.graphviz.org`

```
$ ./reverCSP examples/rc2020.csp
[...]
Current expression:
MAIN

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

```
Current expression:
MAIN
  | MAIN
(P [|{|a|}|] Q)
  | Q
(P [|{|a|}|] a->b->SKIP)
  | P
(R [|{|a|}|] a->(b->SKIP |~| Q) [|{|a|}|] a->b->SKIP)
  | Reverse evaluation
  | Q
(R [|{|a|}|] a->(b->SKIP |~| Q) [|{|a|}|] Q)
  | P
(P [||a||] Q)
```

Fig. 2. Main menu (left) and a series of user actions and the resulting states (right).

erwise, only DOT files will be produced. Both systems are also freely available under open-source licenses.

3.2 Main functionality

reverCSP implements in Erlang a reversible CSP interpreter with two phases:

Generation of tracks. Tracks can be generated using a random number of steps (a random execution) or following the computation steps defined by the user (user-directed execution). This means that, at any point of the computation, the user can choose how to proceed and the associated track is dynamically generated. For instance, a user can perform, say, 50 random steps, then go backward, say 20 steps, and then go forward again but selecting a different rule to be applied. Thus, a different computation (and track) is produced.

Exploration of tracks. Provided that we have a track generated, it can be traversed backward. The traversal is done with computation steps that can be deterministic (using the `Undo` option) or causally-consistent (using the `Reverse evaluation` option). After each step, the system shows the current expression and it gives the option to output the trace and the track. Figure 2 shows the menu displayed during a computation (left), followed by the computation steps selected by the user (right). The states reached are in black, the user actions are in blue and the changes in the state produced by the last action selected are in red.

3.3 Architecture and implementation details

Figure 3 shows the architecture of *reverCSP*. The source code is parsed by module `CSP tracker` to produce an initial state (of the operational semantics). This state is used by module `Forward Computation` to perform a forward step and generate the associated track. If we want to reverse the computation, then module `Backward Computation` can update the state with the information of the

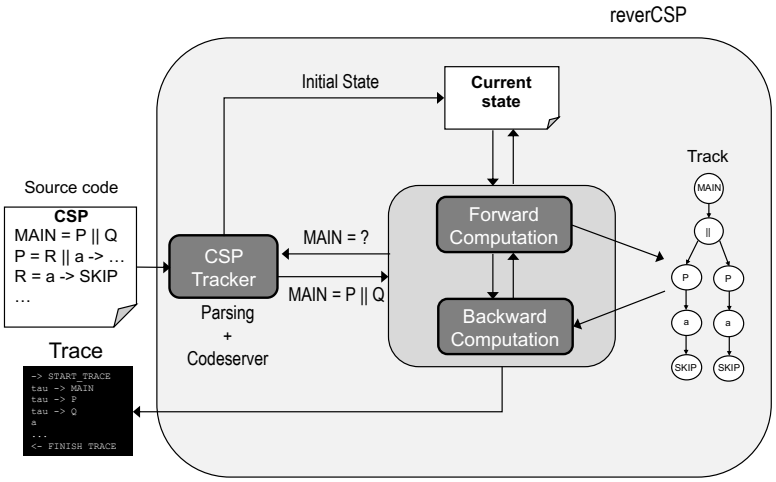

Fig. 3. *reverCSP* architecture.

track. When required, module `CSP tracker` serves the parsed code to the other modules and performs semantic steps from a given state. The interface interacts with the user and continuously displays the trace of the computation.

4 Related Work

There exist different works that propose techniques for rollback-recovery [5] and for reversibility in sequential systems [15] and concurrent systems [11]. Our system, *reverCSP*, is a replay debugger that uses tracks to record the execution. In the core of our tool we use a library called CSP-tracker [13] that can be invoked to produce tracks. One interesting tool that is related to our work is CauDEr [10]. It can also causal-consistently reverse computations, but in this case for Erlang and using a different notion of track. The idea of reversing computations in a causally-consistent manner was introduced in [4] for CCS. Since then, different approaches have emerged. A survey that very nicely describes some of those approaches is [9].

There are other systems such as [1, 2] and [11] that are somehow related to our tool. The work in [1] proposes a modular framework that can be used to define causal-consistent reversible extensions of different concurrent models and languages. The extension of tracks that we defined was inspired by that work. Another interesting work that also proposes a tool that can reverse computations, this time for a CSP-based language embedded in Scala, was presented by Brown and Sabry [2]. Unfortunately, the implementation is not publicly available. Finally, Lanese et al. [11] proposed a novel approach called *controlled causal-consistent replay* where the debugger displays all and only the causes of an error. These approaches are also related to causally-consistent dynamic

slicing [14], but there are important differences: They target pi calculus and we target CSP. Our tool is based on tracks to reverse computations, while dynamic slicing uses execution traces to compute program slices that contain the parts of the source that could influence a given behavior.

5 Conclusions

This paper described *reverCSP*, a tool for the animation and analysis of CSP specifications. On the practical side, *reverCSP* can be seen as a CSP animator with the ability to replay and reverse computations. This ability is provided by the fact that *reverCSP* records every execution step of the computation in a graph-like data structure called track.

We have extended the original definition of track to incorporate timestamps that make explicit the order in which the components of the specification were executed; and this order allows us to reverse the computation. *reverCSP* implements different functionalities such as step-by-step forward and backward execution, random (multiple) steps, undo, and rollback. Besides, it allows to perform both deterministic and causally-consistent reversible steps.

Because *reverCSP* (re)generates the corresponding part of the track with every computation step, the complete track is available to perform different post-mortem analyses. One of them is program slicing, which was already implemented in a tool called CSP-tracker. As future work we plan to adapt our analyses to also implement a causally-consistent dynamic program slicer based on tracks for CSP.

6 Acknowledgements

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, and by the *Generalitat Valenciana* under grant Prometeo/2019/098 (DeepTrust).

References

1. A. Bernadet and I. Lanese. A Modular Formalization of Reversibility for Concurrent Models and Languages. In *Proc. ICE 2016*, EPTCS, 2016.
2. G. Brown and A. Sabry. Reversible communicating processes. *Electronic Proceedings in Theoretical Computer Science*, 203:45–59, 2016.
3. M. Conserva Filhoa, M. Oliveira, A. Sampaio, and A. Cavalcanti. Compositional and local livelock analysis for csp. *Inf. Process. Lett*, 133:21–25, 2018.
4. V. Danos and J. Krivine. Reversible communicating systems. In *Proc. CONCUR'04*, volume 3170 of *LNCS*, pages 292–307. Springer, 2004.
5. 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.*, 34(3):375–408, 2002.

6. Y. Fang, H. Zhu, F. Zeyda, and Y. Fei. Modeling and analysis of the disruptor framework in csp. In *Proc. CCWC'18*. IEEE Computer Society, 2018.
7. P. B. Ladkin and B. B. Simons. Static deadlock analysis for CSP-type communications. In *Responsive Computer Systems: Steps Toward Fault-Tolerant Real-Time Systems. The Springer International Series in Engineering and Computer Science, vol 297. Springer*, 1995.
8. R. Landauer. Irreversibility and heat generation in the computing process. *IBM J. Res. Dev.*, 5:183–191, 1961.
9. I. Lanese, C. Antares Mezzina, and F. Tiezzi. Causal-consistent reversibility. *Bulletin of the EATCS*, 114:17, 2014.
10. I. Lanese, N. Nishida, A. Palacios, and G. Vidal. CauDEr: A causal-consistent reversible debugger for erlang. In *Proc. FLOPS 2018*, volume 10818 of *LNCS*, pages 247–263, 2018.
11. I. Lanese, A. Palacios, and G. Vidal. Causal-consistent replay debugging for message passing programs. In *Proc. FORTE 2019*, pages 167–184, June 2019.
12. M. Llorens, J. Oliver, J. Silva, and S. Tamarit. Dynamic slicing of concurrent specification languages. *Parallel Computing*, 53:1–22, 2016.
13. M. Llorens, J. Oliver, J. Silva, and S. Tamarit. Tracking CSP computations. *J. Log. Algebr. Meth. Program.*, 102:138–175, 2019.
14. R. Perera, D. Garg, and J. Cheney. Causally consistent dynamic slicing. In *Proc. CONCUR'16*, volume 59 of *LIPIcs*, pages 18:1–18:15, 2016.
15. I. Phillips, I. Ulidowski, and S. Yuen. A reversible process calculus and the modelling of the erk signalling pathway. In *Proc. RC'12*, volume 7581 of *LNCS*, pages 218–232. Springer, 2012.
16. A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.
17. H. Zhao, H. Zhu, F. Yucheng, and L. Xiao. Modeling and verifying storm using csp. In *Proc. HASE'19*. IEEE Computer Society, 2019.