Implementation of a Reversible Distributed Calculus

Clément Aubert, Peter Browning

Implementation of Reversible Distributed Calculus (tool demonstration paper)

Clément Aubert[0000−0001−6346−3043] and Peter Browning

School of Computer & Cyber Sciences, Augusta University, Augusta, USA
{caubert, pebrowning}@augusta.edu

Abstract Process calculi (π-calculus, CCS, ambient calculus, etc.) are an abstraction of concurrent systems useful to study, specify and verify distributed programs and protocols. This project, IRDC, is concerned with the implementation of such an abstraction for *reversible* process calculi. It is, to the best of our knowledge, the first such publicly available tool. We briefly present the current state of this tool, some of its features, and discuss its future developments.

Keywords: Formal semantics · Process algebra and calculi · Reversible Computation · Concurrency · Tool implementation

1 Implementations of (Reversible) Concurrent Calculi

Implementing a process calculus such as π-calculus, CCS or the ambient calculus serves four overlapping goals:

- It allows to machine-check theorems and definitions [**?**,**?**,**?**] using proof assistants such as Coq [**?**], resulting sometimes in simplification [**?**] or the finding of regrettable imprecision or errors [**?**].
- Using it as an actual programming language, it enables the implementation of toy programs [**?**] that exemplifies the purpose and expressivity of the calculus.
- It can also be used as a specification language: typically, the Proverif tool [**?**], which implements the applied π-calculus [**?**], has been used to certify and model security protocols in a variety of areas [**?**].
- Last but not least, it serves a pedagogical purpose. Fleshing out the abstract theory lets users and programmers get acquainted with such systems, and showcases its interest. Simply for CCS, projects from the Sapienza Università di Roma, the Università di Bologna, the Università di Pisa or the Aalborg Universitet [**?**] demonstrate a vivid interest for implementations using a variety of languages (Standard ML, SWI-Prolog or Typescript) and approaches (ability to "play the bisimulation game", focus on program's correctness, etc.).

Reversible process calculi emerged almost 20 years ago with Reversible CCS (RCCS) [**?**] and CCS with keys (CCSK) [**?**]. Both calculi evolved over the time, and brought many interesting insights both on reversibility and on concurrency. However, aside from SimCCSK [**?**]–which is not publicly available and

not maintained since 2008 to our knowledge–no implementation of concurrent, reversible calculus exists. This short paper intends to present the current status of an implementation of CCSK, called *Implementation of Reversible Distributed Calculus* (IRDC), we have been working on since February 2022. While it is still a work-in-progress, we believe that gathering early feedback and showcasing some of the challenges we faced and solved will be of interest to the community.

2 System Design

We chose to implement CCSK, a variation on CCS that attaches keys to past action instead of discarding them, and in which every derivation rule has an inverse. Our syntax is faithful to recent papers on the topic [**?**]:

`0`	Nil process	`a`, ..., `z`	Input channel name
`+`	Summation operator	`'a`, ..., `'z`	Output channel name
`|`	Parallel operator	`Tau{'a, a}`	Silent action on `a`
`\{a, b}`	Restriction along `a`, `b`	`a[k0]`, ..., `'z[k1]`	Label keys

Details can be found in our readme's "Syntax and Precedence of Operators". The operational semantics is standard, reminded in our documentation, and can be partially inferred from the following:

```
                          ------| Actionable Labels |------
                          [0] b
(((a+b)|'b)|'a)\{a}       [1] 'b
                          [2] Tau{'b, b}
                          [3] Tau{'a, a}
```

After performing e.g., a synchronization on `a` (action `[3]`) and performing an output on `'b` (action `[1]`), this process would become

```
(((Tau{'a, a}[k1].0+b)|'b[k3].0)|Tau{'a, a}[k1].0)\{a}
```

from which the actions with keys `k1` and `k3` could both be undone, in any order. Note that we decided to omit the nil process when preceded by an actionable prefix,[1] and that we label the `Tau` silent action with the name of the complementary channels that synchronized.[2]

3 Running and Installing IRDC

Downloading the latest release can done by browsing to `https://github.com/CinRC/IRDC-CCSK/releases` or by executing this simple one-liner[3]:

```
curl -s https://api.github.com/repos/CinRC/IRDC-CCSK/releases/latest \
| grep browser_download_url | cut -d : -f 2,3 | tr -d \" | wget -qi -
```

Listing 3.1. Fetching the latest release in one command

[1] This behavior can be toggled with the `--require-explicit-null` option.

[2] As discussed at the end of **??**, this does not impact our implementation of simulation, as e.g., `Tau{'a, a}` and `Tau{'b, b}` are treated as equivalent.

[3] The source code is also archived at `https://archive.softwareheritage.org/browse/origin/directory/?origin_url=https://github.com/CinRC/IRDC-CCSK`.

The process presented in **??** can then be executed using

```
java -jar IRDC-*.jar "(((a+b)|'b)|'a)\{a}"
```

Listing 3.2. Launching the command-line interface with a demo process

The user can then decide on which channel name (for forward transition) or label key (for backward transition) they want to act by selecting the corresponding number and hitting ⏎. As of now, the program needs to be killed manually (for instance using ctrl+c on Unix systems) to be exited. Many flags can be used to tweak the program's behavior or its output. They can be accessed using the `--help` flag or by consulting our readme. We present some of them below.

4 Features

As of now, our tool's primary goal is pedagogical, to raise interest and train in distributed, reversible computation. The user can interact with a process to get a better understanding of the operators' mechanisms and semantics. They can for instance try to answer the question

> Can the process `(a.b.c|('a.'c+'a.'b))\{a,c}` synchronize on `c`, and if yes, using which reduction sequence?

using a command such as

```
java -jar IRDC-*.jar --dL "(a.b.c|('a.'c+'a.'b))\{a,c}"
```

Listing 4.1. Process example to understand restriction and prefixing

where the `--dL` flag helps by numbering the occurrences of channel names.

It is also possible to check one's answer using the `--enumerate` flag, that lists all possible (forward-only) execution sequences:

```
java -jar IRDC-*.jar --enumerate "(a.b.c|('a.'c+'a.'b))\{a,c}"
(a.b.c|('a.'c+'a.'b))\{a,c}
├─Tau{'a, a}─ (Tau{'a, a}[k0].b.c|(Tau{'a, a}[k0].'c+'a.'b))\{a,c}
│ └─b─ (Tau{'a, a}[k0].b[k2].c|(Tau{'a, a}[k0].'c+'a.'b))\{a,c}
│ └─Tau{'c, c}─ (Tau{'a, a}[k0].b[k2].Tau{'c, c}[k3].0|(Tau{'a, a}[k0].Tau{'c,
    ↪ c}[k3].0+'a.'b))\{a,c}
└─Tau{'a, a}─ (Tau{'a, a}[k1].b.c|('a.'c+Tau{'a, a}[k1].'b))\{a,c}
  ├─b─ (Tau{'a, a}[k1].b[k5].c|('a.'c+Tau{'a, a}[k1].'b))\{a,c}
  │ └─'b─ (Tau{'a, a}[k1].b[k5].c|('a.'c+Tau{'a, a}[k1].'b[k7].0))\{a,c}
  ├─'b─ (Tau{'a, a}[k1].b.c|('a.'c+Tau{'a, a}[k1].'b[k6].0))\{a,c}
  │ └─b─ (Tau{'a, a}[k1].b[k8].c|('a.'c+Tau{'a, a}[k1].'b[k6].0))\{a,c}
  └─Tau{'b, b}─ (Tau{'a, a}[k1].Tau{'b, b}[k4].c|('a.'c+Tau{'a, a}[k1].Tau{'b,
    ↪ b}[k4].0))\{a,c}
```

Listing 4.2. Demonstrating the `--enumerate` flag

Our program can also be used to exhibit the difference between temporal and causal orders in reversible systems. For instance, the following execution:

```
java -jar IRDC-*.jar "a.b | 'b | c"
((a.b|'b)|c) -c-> ((a.b|'b)|c[k0].0)
 -a-> ((a[k1].b|'b)|c[k0].0)
 -Tau{'b, b}-> ((a[k1].Tau{'b, b}[k3].0|Tau{'b, b}[k3].0)|c[k0].0)
 ~[k0]~> ((a[k1].Tau{'b, b}[k3].0|Tau{'b, b}[k3].0)|c)
```

Listing 4.3. Illustrating the difference between temporal and causal orders

makes it clear that even if the action on `c` (to which the key `k0` was given) was triggered first when performing forward transitions, it can be undone first.

Traditional (e.g., forward-only) simulation was also implemented, but cannot be accessed interactively. Our `SimulationTest` class, however, asserts, e.g.,

- that `a.(b+c)` can simulate `a.b + a.c` but cannot be simulated by it,
- that `(a|'a)\{a}` and `(b|'b)\{b}` are equivalent,
- that `a|b` can simulate `a.b` but cannot be simulated by it, etc.

5 Inner Flow

Our program works in three stages. The `CCSParser` class initially traverses the input and tokenizes each node (i.e., operand) of a process. Each node is then instantiated as an object internally. Then, the program recursively links the nodes together using the appropriate order of binding power of the operators. After the nodes are all linked, only one process is expected to remain: this process is the "ancestor" that can then be acted on.

Internally, the program does not de-allocate keys when they are freed, so that a process going constantly back-and-forth will always receive different keys. However, our `NodeIDGenerator` class and its `.nextAvailableKey()` method are modular enough that alternative formalisms could be easily implemented.

6 Software Engineering Best Practices

Our implementation strives to use state-of-the-art technologies and to promote best practices. We decided to use to use the Java Development Kit (17.0.2) for its flexibility, portability and popularity.[4] We use the software project management and comprehension tool Maven (3.6.3) to ease adoption and dependency management, but also to enforce stylistic constraints: our implementation enforces the `google_checks.xml` checkstyle guideline for uniformity and best practises. Our code is versioned using git, publicly available, open source and uses semantic versioning to ease collaboration and adoption.

We also leveraged github's Continuous integration (CI) / Continuous deployment (CD) to remotely compile our source code and automatically offer pre-compiled releases that can be executed directly. Last but not least, our extensive testing suite makes sure that our implementations of action, enumeration (that lists all possible actions) and restriction, but also our parser and simulation implementations, behave as expected.

7 Road Map

Our list of issues highlights some of our challenges and goals. Among them, developing a graphical user interface is the focus of a capstone project currently taking place at Augusta University.

[4] It should be noted that most existing implementations uses declarative programming languages, and that with that respect our implementation is quite original.

Some of the other milestones include:

- Accepting processes that already started their execution, so that one could use as an input process e.g., `((a.b|'b)|c[k4].0)`,
- as a follow-up, our implementation could decide if a process that already started their execution is "reachable", that is, can be accessed using forward-only transitions,
- and, of course, performing additional tests.

Our most exciting, but also longer-term, challenges, would be to implement one of the "reversible" bisimulation [?,?], a mechanism to distribute the generation of keys [?], or to formally verify the correctness of our definition of concurrency [?].

8 Conclusion

The creation of this tool highlighted some interesting components of CCSK as a whole. Acting much like a calculator, our tool had to create multiple algorithms for manipulating and equivocating different processes. One of the most interesting problems raised by the construction of this tool–and that is not solved yet–is to design, for the first time to our knowledge, an algorithm to determine whether two reversible processes are in an history-preserving bisimulation [?,?,?].

Acknowledgments

The authors would like to thank Brett Williams and John Yalch for their contribution to this project, and Jason Orlosky for his comments on this submission. This material is based upon work supported by the National Science Foundation under Grant No. 2242786 (SHF:Small:Concurrency In Reversible Computations).