Reversible Communicating Processes

Geoffrey Brown Amr Sabry
School of Informatics and Computing
Indiana University, Bloomington, IN
geobrown@indiana.edu sabry@indiana.edu

Reversible distributed programs have the ability to abort unproductive computation paths and backtrack, while unwinding communication that occurred in the aborted paths. While it is natural to assume that reversibility implies full state recovery (as with traditional roll-back recovery protocols), an interesting alternative is to separate backtracking from local state recovery. For example, such a model could be used to create complex transactions out of nested compensable transactions where a programmer-supplied compensation defines the work required to "unwind" a transaction.

Reversible distributed computing has received considerable theoretical attention, but little reduction to practice; the few published implementations of languages supporting reversibility depend upon a high degree of central control. The objective of this paper is to demonstrate that a practical reversible distributed language can be efficiently implemented in a fully distributed manner.

We discuss such a language, supporting CSP-style synchronous communication, embedded in Scala. While this language provided the motivation for the work described in this paper, our focus is upon the distributed implementation. In particular, we demonstrate that a "high-level" semantic model can be implemented using a simple point-to-point protocol.

1 Introduction

Speculative execution either by intent or through misfortune (in response to error conditions) is pervasive in system design and yet it remains difficult to handle at the program level [7]. Indeed, we find that despite the importance of speculative computation, there is very little programmatic support for it in distributed languages at the foundational level it deserves. We note that, from a programming language perspective, speculative execution requires a backtracking mechanism and that, even in the sequential case, backtracking in the presence of various computational effects (e.g. assignments, exceptions, etc.) has significant subtleties [9]. The introduction of concurrency additionally requires a "distributed backtracking" algorithm that must "undo" the effects of any communication events that occurred in the scope over which we wish to backtrack. While this has been successfully accomplished at the algorithmic level (e.g. in virtual time based simulation [12, 11]), in models of concurrent languages (e.g. [2, 3, 13, 14, 21]) and in some restricted parallel shared-memory environments (e.g. [23, 8, 15, 22, 17]), it does not appear that any concurrent languages based upon message passing have directly supported backtracking with no restrictions. The language constructs we introduce are inspired by the stabilizers of [24]; however, that work depends upon central control to manage backtracking. Our work was also inspired by the work of Hoare and others [16, 10]. Communicating message transactions [23, 15] is an interesting related approach that relies upon global shared data structures.

The work presented in this paper has a natural relationship to the rich history of rollback-recovery protocols [5]. Rollback-recovery protocols were developed to handle the (presumably rare) situation where a processor fails and it is necessary to restart a computation from a previously saved state. The fundamental requirement of these protocols is that the behavior is as if no error ever occurred. In contrast, we are interested in systems where backtracking might take the computation in a new direction based upon state information gleaned from an abandoned execution path; the (possibly frequent) decision to

S. J. Gay and J. Alglave (Eds.): Programming Language Approaches to
Concurrency- and Communication-Centric Software (PLACES 2015).
EPTCS 203, 2016, pp. 45–59, doi:10.4204/EPTCS.203.4

backtrack is entirely under program control. Because check-pointing in traditional rollback-recovery protocols involves saving a complete snapshot of a process's state, it is a relatively expensive operation. Much of the research in rollback-recovery protocols focuses upon minimizing these costs. The cost of check-pointing is much lower for our domain – saving control state is no more expensive than for a conventional exception handler; the amount of data state preserved is program dependent.

Implementing a reversible concurrent language is not a trivial undertaking and, as we found, there are many opportunities to introduce subtle errors. Ideally, such a language implementation should be accompanied by a suitable semantics that provides both a high-level view which a programmer can use to understand the expected behavior of a program text, and a low-level view which the language implementer can use to develop a correct implementation. In order to accommodate these two constituencies, we have developed two separate semantic models. We have developed a refinement mapping to demonstrate that the low-level model is a correct implementation of the high-level model.

The remainder of this paper is organized as follows. We begin with a small example that illustrates the main ideas using our Scala implementation. We then, in Sec. 3, present a formal "high-level" semantic model focusing on the semantics of forward communication and backtracking. In Sec. 4, we discuss a communication protocol for maintaining backtracking state across distributed communicating agents. Sec. 5 introduces a "low-level" model that utilizes the channel protocol to implement the high-level model, and outlines the proof of correctness of the low-level semantics with respect to the high-level semantics. We end with a brief discussion.

2 Example

A programmer wishing to use our distributed reversible extensions of Scala imports our libraries for processes and channels and then defines extensions of the base class `CspProc` by overriding the method `uCode`. The user-defined code must use our channel implementation for communication and may additionally use the keywords `stable` and `backtrack` for managing backtracking over speculative executions. The `stable` regions denote the scope of saved contexts; executing `backtrack` within a stable region returns control to the beginning of the stable region – much like throwing an exception, but with the additional effect of unwinding any communication that may have occurred within the stable region.

The excerpt of Fig. 1 provides the code for two processes `p1` and `p2` that communicate over channel `c` – not shown is the code that creates these processes and the channel. Process `p1`'s execution consists of entering a stable region, entering a nested stable region (line 6), sending a message to `p2` (line 8), entering another stable region (line 9), sending another message to `p2` (line 11), and then possibly backtracking (line 15). Meanwhile process `p2` also starts a stable region (line 24) in which it receives the two messages (lines 26 and 29).

A possible execution trace showing a possible interleaving of the execution is:

```
p2: start
p1: start
p1: snd 2
p2: recv 2
p1: snd 2
p2: recv 2
p1: backtrack
p2: start
p1: snd 1
p2: recv 1
p1: snd 1
p2: recv 1
```

```
   class p1 (c: SndPort, name: String)
2      extends CspProc(name) {
     override def uCode = {
4      println("p1: start")
       var count = 2
6      stable {
         println("p1: snd " + count)
8        send(c,count)
         stable {
10         println("p1: snd " + count)
           send(c,count)
12         count = count - 1
           if (count > 0) {
14           println ("p1: backtrack")
             backtrack }
16       }}
     }}
18
```

```
   class p2 (c: RcvPort, name: String)
20     extends CspProc(name) {
     override def uCode = {
22     println("p2: start")
       stable {
24       var x = receive(c)
         println("p2: recv " + x)
26       var y = receive(c)
         println("p2: recv " + y)
28     }
     }}
30
   class RootClass() extends CspProc("root") {
32   override def uCode() = {
       stable{
34       var c = newChannel("c")
         par (new p1(c.tx, "p1"),
36            new p2(c.rx, "p2"))
```

Figure 1: Example

In the trace, processes `p1` and `p2` both start executing and enter their respective stable regions. Process `p2` must block until a communication event occurs on channel `c`. Process `p1` initiates the communication sending the value `2` which is received by `p2` which must then block again waiting to receive on channel `c`. Process `p1` enters its nested stable region and sends another value `2` which is received. At this point, process `p2` is "done" but process `p1` decides to backtrack. As a result, process `p1` transfers its control to the inner stable region (line 9). This jump invalidates the communication on channel `c` at line 11. Process `p1` then blocks until process `p2` takes action. When process `p2` notices that the second communication event within the stable region (line 29) was invalidated, it backtracks to the start of its stable region. This jump invalidates the first communication action (line 26) which in turn invalidates the corresponding action in `p1` at line 8. In other words, process `p1` is forced to backtrack to its outer stable region to establish a causally consistent state. It is important to remember that all processes have an implicit stable region that includes their full code body; `p2` is therefore forced to backtrack to the beginning of its code. In other words, after the backtracking of `p1`, both communication events between `p1` and `p2` are re-executed.

3 High-Level Semantics of a Reversible Process Language

We will present two semantic models for our language. The first "high-level" semantics formalizes both the forward communication events that occur under "normal" program execution and the backwards communication events that occur when processes are backtracking to previously saved states as atomic steps. In the low-level semantics, these communication events are further subdivided into actions that communicating senders and receivers may take independently in a distributed environment and hence trades additional complexity for a specification that is close to a direct implementation. This low-level semantics is based upon a channel protocol that we have verified using the SAL infinite-state model checker [19, 20, 4]; the invariants validated using SAL were necessary to prove that the low-level semantics is a refinement of the high level semantics. Both of our semantic models are based upon virtual time – a commonly used technique for conventional rollback recovery protocols [6, 18]. Our approach differs in utilizing synchronous communication and also by providing a fully distributed rollback protocol.

3.1 User-Level Syntax

We begin with a core calculus which is rich enough to express the semantic notions of interest:

$$\begin{array}{llll}
(\textit{channel names}) & \ell & & \\
(\textit{constants}) & c & ::= & () \mid 0 \mid 1 \mid \ldots \mid + \mid - \mid \geq \mid \ldots \\
(\textit{expressions}) & e & ::= & c \mid x \mid \lambda x.e \mid e_1 e_2 \mid \mathsf{send}\ \ell\ e \mid \mathsf{recv}\ \ell(x).e \mid \mathsf{stable}\ e \mid \mathsf{backtrack}\ e \\
(\textit{processes}) & p & ::= & p_1 \parallel p_2 \quad \mid \quad \langle e \rangle
\end{array}$$

A program is a collection of processes executing in parallel. Expressions extend the call-by-value λ-calculus with communication and backtracking primitives. The communication primitives are $\mathsf{send}\ \ell\ e$ which commits to sending the value of e on the channel ℓ and $\mathsf{recv}\ \ell(x).e$ which blocks until it receives x on channel ℓ. Our Scala implementation supports input "choice" allowing a receiving process to non-deterministically choose among a collection of active channels; the addition of choice is necessary for expressiveness but adds little new insight to the formal semantics and is hence omitted in the interest of brevity. The backtracking primitives are $\mathsf{stable}\ e$ which is used to delimit the scope of possible backtracking events within e. The expression $\mathsf{backtrack}\ e$ typically has two effects: the control state in the process executing the instruction jumps back to the dynamically closest nested block with the value of *e and* all intervening communication events are invalidated. The latter action might force neighboring processes to also backtrack, possibly resulting in a cascade of backtracking for a poorly written program.

3.2 Internal Syntax

In order to formalize evaluation, we define a few auxiliary syntactic categories that are used to model run-time data structures and internal states used by the distributed reversible protocol. These additional categories include process names, time stamps, channel maps, evaluation contexts, and stacks:

$$\begin{array}{llll}
(\textit{process names}) & n & & \\
(\textit{time stamps}) & t & & \\
(\textit{values}) & v & ::= & c \mid x \mid \lambda x.e \mid \mathsf{stable}\ (\lambda x.e) \\
(\textit{expressions}) & e & ::= & \ldots \mid \underline{\mathsf{stable}}\ e \\
(\textit{evaluation contexts}) & E & ::= & \square \mid E\ e \mid v\ E \mid \mathsf{send}\ \ell\ E \mid \mathsf{stable}\ E \mid \underline{\mathsf{stable}}\ E \mid \mathsf{backtrack}\ E \\
(\textit{channel maps}) & \Xi & = & \ell \mapsto (n,t,n) \\
(\textit{stacks}) & \Gamma & = & \bullet \mid \Gamma,(E,v,t,\Xi) \\
(\textit{processes}) & p & ::= & p_1 \parallel p_2 \quad \mid \quad \langle n@t:\ \Gamma, e \rangle \\
(\textit{configurations}) & C & ::= & \Xi \fatsemi (p_1 \parallel p_2\ \ldots\ \parallel p_k)
\end{array}$$

Expressions are extended with $\underline{\mathsf{stable}}\ e$ which indicates an *active* region. The syntax of processes $\langle n@t:\ \Gamma, e\rangle$ is extended to record additional information: a process id n, a virtual time t, a context stack Γ, and an expression e to evaluate. The processes communicate using channels ℓ whose state is maintained in maps Ξ. Each entry in Ξ maps a channel to the sender and receiver processes (which are fixed throughout the lifetime of the channel) and the current virtual time of the channel. Contexts are pushed on the stack when a process enters a new stable region and popped when a process backtracks or exits a stable region. Each context includes a conventional continuation (modeled by an evaluation context E), a value v with which to backtrack if needed, a time stamp, and a local channel map describing the state of the communication channels at the time of the checkpoint.

(H1) $\Xi \fatsemi \langle n@t : \Gamma, E[(\lambda x.e)\ v]\rangle \xrightarrow{\varepsilon} \Xi \fatsemi \langle n@t : \Gamma, E[e[v/x]]\rangle$

(H2) $\Xi\{\ell \mapsto (n_1, t_c, n_2)\} \fatsemi \langle n_1@t_1 : \Gamma_1, E_1[\mathsf{send}\ \ell\ v]\rangle \parallel \langle n_2@t_2 : \Gamma_2, E_2[\mathsf{recv}\ \ell(x).e]\rangle \xrightarrow{\ell@t[v]}$
$\Xi\{\ell \mapsto (n_1, t, n_2)\} \fatsemi \langle n_1@t : \Gamma_1, E_1[()]\rangle \parallel \langle n_2@t : \Gamma_2, E_2[e[v/x]]\rangle$
where $t > \max(t_1, t_2)$

(H3) $\Xi \fatsemi \langle n@t : \Gamma, E[(\mathsf{stable}\ (\lambda x.e))\ v]\rangle \xrightarrow{\varepsilon}$
$\Xi \fatsemi \langle n@t' : \Gamma, (E[(\mathsf{stable}\ (\lambda x.e))\ \square], v, t, \Xi_n), E[\underline{\mathsf{stable}}\ e[v/x]]\rangle$
where $t' > t$ and Ξ_n is the subset of Ξ referring to the channels of n

(H4) $\Xi \fatsemi \langle n@t : \Gamma, (E', v', t', \Xi'), E[\underline{\mathsf{stable}}\ v]\rangle \xrightarrow{\varepsilon} \Xi \fatsemi \langle n@t : \Gamma, E[v]\rangle$

(H5) $\Xi \fatsemi \langle n@t : \Gamma, (E', v', t', e'), e\rangle \xrightarrow{\varepsilon} \Xi \fatsemi \langle n@t : \Gamma, (E', v', t', e'), \mathsf{backtrack}\ v'\rangle$

(H6) $\Xi\{\ell \mapsto (c_1, t_c, c_2)\} \fatsemi \langle n_1@t_1 : \Gamma_1, E_1[\mathsf{backtrack}\ v_1]\rangle \parallel \langle n_2@t_2 : \Gamma_2, E_2[\mathsf{backtrack}\ v_2]\rangle \xrightarrow{\bar{\ell}@t'_c}$
$\Xi\{\ell \mapsto (c_1, t'_c, c_2)\} \fatsemi \langle n_1@t_1 : \Gamma_1, E_1[\mathsf{backtrack}\ v_1]\rangle \parallel \langle n_2@t_2 : \Gamma_2, E_2[\mathsf{backtrack}\ v_2]\rangle$
where $0 \le t'_c < t_c$ and $\{c_1, c_2\} = \{n_1, n_2\}$

(H7) $\Xi\{\ell \mapsto (-, t, -)\} \fatsemi \langle n_1@t_1 : \Gamma, (E_1, v', t'_1, \Xi_1\{\ell \mapsto (-, t', -)\}), E[\mathsf{backtrack}\ v]\rangle \xrightarrow{\varepsilon}$
$\Xi\{\ell \mapsto (-, t, -)\} \fatsemi \langle n_1@t_1 : \Gamma, E[\mathsf{backtrack}\ v]\rangle$
where $t < t'$

(H8) $\Xi \fatsemi \langle n_1@t_1 : \Gamma, (E_1, v_1, t'_1, \Xi_1), E[\mathsf{backtrack}\ v]\rangle \xrightarrow{\varepsilon} \Xi \fatsemi \langle n_1@t'_1 : \Gamma, E_1[v]\rangle$
where the timestamp on every channel in Ξ_1 is equal to the timestamp of the same channel in Ξ

Figure 2: High-level rules

A semantic configuration C consists of a global channel map Ξ and a number of processes. An invariant maintained by the semantics is that a process executing in the forward direction will have the times of its channels in the global map greater than or equal to the times associated with the channels in the top stack frame. Similarly, the time associated with a process will always be at least as great as the times associated with its channels. Both invariants follow from the intuition that any channel appearing on the stack must have been pushed "in the past" and similarly that any communication reflected in the global map must have also happened "in the past." A process in the backtracking state will temporarily violate these invariants until it negotiates a consistent state with its neighbors.

We assume that in the initial system state, all channels have time 0 and every process is of the form $\langle n@0 : \bullet, \mathsf{stable}\ (\lambda\_.e)\ ()\rangle$; i.e. process n is entering a stable region containing the expression e with an empty context stack at time 0.

3.3 Forward Semantics

The rules are collected in Fig. 2. We discuss each rule with the aid of simple examples below.

A computation step that does not involve communication, stable regions, or backtracking is considered a local computation step. None of the internal structures need to be consulted or updated during such local computation steps and hence in our Scala implementation, local computation steps proceed at "full native speed." In order to establish notation, here is a simple computation step:[1]

$$\{\ell \mapsto (n_1, 2, n_2)\} \fatsemi \langle n_1@5 : \Gamma, 1+2\rangle \rightarrow \{\ell \mapsto (n_1, 2, n_2)\} \fatsemi \langle n_1@5 : \Gamma, 3\rangle$$

[1]The color version of the paper highlights the components of the configuration that are modified by each rule.

In this example, a process n_1 with local virtual time 5 and making forward progress encounters the computation $1+2$. We assume the existence of a channel ℓ which is associated with time 2 and connects n_1 to n_2. Intuitively this means that the last communication by that process on that channel happened three virtual time units in the past. The reduction rule leaves all structures intact and simply performs the local calculation. In the general case, we have rule H1 for application of λ-expressions and similar rules for applying primitive operations. In rule H1, the runnable expression in the process is decomposed into an evaluation context E and a current "instruction" $(\lambda x.e)\ v$. This instruction is performed in one step that replaces the parameter x with the value v in the body of the procedure e. The notation for this substitution is $e[v/x]$. The entire transition is tagged with ε indicating that it produces no visible events.

Communication between processes is synchronous and involves a handshake. We require that in addition to the usual exchange of information between sender and receiver, that the handshake additionally exchanges several virtual times to force the virtual times of the sending process, the receiving process, and the used channel to be all equal to a new virtual time larger than any of the prior times for these structures. Here is a small example illustrating this communication handshake:

$$\begin{array}{ll} & \{\ell \mapsto (n_1, 3, n_2)\} ⨟ \langle n_1 @ 5 : \Gamma, \mathsf{send}\ \ell\ 10\rangle \parallel \langle n_2 @ 4 : \Gamma, \mathsf{recv}\ \ell(x).x+1\rangle \\ \to & \{\ell \mapsto (n_1, 6, n_2)\} ⨟ \langle n_1 @ 6 : \Gamma, ()\rangle \parallel \langle n_2 @ 6 : \Gamma, 10+1\rangle \end{array}$$

Initially, we have two processes willing to communicate on channel ℓ. Process n_1 is sending the value 10 and process n_2 is willing to receive an x on channel ℓ and proceed with $x+1$. After the reduction, the value 10 is exchanged and each process proceeds to the next step. In addition, the virtual times of the two processes as well as the virtual time of the channel ℓ have all been synchronized to time 6 which is greater than any of the previous times. This is captured in rule H2. In that rule, the fact that $t > t_c$ follows from the global model invariant that $t_1 \geq t_c \wedge t_2 \geq t_c$. The notation $\Xi\{\ell \mapsto (n_1, t_c, n_2)\}$ says that, in channel map Ξ, channel ℓ connects sender n_1 and receiver n_2 and has time t_c. The transition produces the visible event $\ell @ t[v]$ that value v was transferred on channel ℓ at time t.

Finally, there are rules H3, H4, and H5 corresponding to entering and exiting stable regions. These rules are interconnected and we illustrate them with a small example whose first transition is:

$$\begin{array}{ll} & \{\ell \mapsto (n_1, 2, n_2)\} ⨟ \langle n_1 @ 5 : \Gamma, 7 + (\mathsf{stable}\ f)\ v\rangle \\ \to & \{\ell \mapsto (n_1, 2, n_2)\} ⨟ \langle n_1 @ 6 : \Gamma(7+(\mathsf{stable}\ f)\ \square, v, 5, \{\ell \mapsto (n_1, 2, n_2)\}), 7 + \underline{\mathsf{stable}}\ (f\ v)\rangle \end{array}$$

Process n_1 encounters the expression $7+(\mathsf{stable}\ f)\ v$ where f is some function and v is some value. The stable construct indicates that execution might have to revert back to the current state if any backtracking actions are encountered during the execution of $f\ v$. The first step is to increase the virtual time to establish a new unique event. Then, to be prepared for the eventuality of backtracking, process n_1 pushes $(7+(\mathsf{stable}\ f)\ \square, v, 5, \{\ell \mapsto (n_1, 2, n_2)\})$ on its context stack. The pushed information consists of the continuation $7+\mathsf{stable}\ f\ \square$ which indicates the local control point to jump back to, the value v, the virtual time 5 which indicates the time to which to return, and the current channel map which captures the state of the communication channels to be restored. Execution continues with $7+\underline{\mathsf{stable}}\ (f\ v)$ where the underline indicates that the region is currently active. If the execution of $f\ v$ finishes normally, for example, by performing communication on channel ℓ and then returning the value 100, then the evaluation progresses as follows:

$$\begin{array}{ll} & \{\ell \mapsto (n_1, 8, n_2)\} ⨟ \langle n_1 @ 8 : \Gamma, (7+(\mathsf{stable}\ f)\ \square, v, 5, \{\ell \mapsto (n_1, 2, n_2)\}), 7 + \underline{\mathsf{stable}}\ 100\rangle \\ \to & \{\ell \mapsto (n_1, 8, n_2)\} ⨟ \langle n_1 @ 8 : \Gamma, 7 + 100\rangle \end{array}$$

The context stack is popped and execution continues in the forward direction. The case in which $f\ v$ backtracks is considered in the next section.

3.4 Backtracking Semantics

Backtracking may occur either from within the current process or indirectly because another neighboring process has retracted a communication event. In the first case, the saved context will be resumed with a value of the programmer's choice; in the latter case, the context will be resumed, asynchronously, with the value saved on the context stack. A well-typed program should have the function argument to `stable` ($\lambda x.e$ in the rules above) be prepared to handle either situation.

We illustrate the important steps taken in a typical backtracking sequence using an example. Consider the following configuration in which process n_1 has communicated on channel ℓ at time 2, taken a step to time 3, entered a stable region, communicated again on ℓ at time 5, taken two steps to time 7, entered another stable region, communicated again on ℓ at time 8, taken a step to time 9, and then encountered a backtracking instruction. The internal state of its communicating partner is irrelevant for the example except that its virtual time is assumed to be larger than 8:

$$\begin{array}{l}\{\ell \mapsto (n_1, 8, n_2)\} \,⨾\, \langle n_1@9 : \Gamma, (E_1[(\mathsf{stable}\ f_1)\ \square], v_1, 3, \{\ell \mapsto (n_1, 2, n_2)\}) \\ \qquad\qquad\qquad\qquad , (E_2[(\mathsf{stable}\ f_2)\ \square], v_2, 7, \{\ell \mapsto (n_1, 5, n_2)\}), \\ \qquad\qquad E_3[\mathsf{backtrack}\ 100]\rangle \\ \qquad\quad \parallel \langle n_2@13 : \Gamma_2, e_2\rangle\end{array}$$

The first step is for two processes to negotiate a time to which channel ℓ should return. The semantic specification is flexible allowing *any* time in the past (including 0 in the extreme case). For the running example, we pick time 2 for the channel ℓ. The configuration steps to:

$$\begin{array}{l}\{\ell \mapsto (n_1, 2, n_2)\} \,⨾\, \langle n_1@9 : \Gamma, (E_1[(\mathsf{stable}\ f_1)\ \square], v_1, 3, \{\ell \mapsto (n_1, 2, n_2)\}) \\ \qquad\qquad\qquad\qquad , (E_2[(\mathsf{stable}\ f_2)\ \square], v_2, 7, \{\ell \mapsto (n_1, 5, n_2)\}), \\ \qquad\qquad E_3[\mathsf{backtrack}\ 100]\rangle \\ \qquad\quad \parallel \langle n_2@13 : \Gamma_2, e_2\rangle\end{array}$$

At this point, neither process may engage in any forward steps. Focusing on n_1 for the remaining of the discussion, the top stack frame needs to be popped as its embedded time for channel ℓ is *later* than the global time:

$$\{\ell \mapsto (n_1, 2, n_2)\} \,⨾\, \langle n_1@9 : \Gamma, (E_1[(\mathsf{stable}\ f_1)\ \square], v_1, 3, \{\ell \mapsto (n_1, 2, n_2)\}), E_3[\mathsf{backtrack}\ 100]\rangle$$

In general, the popping of stack frames continues until the time associated with all the channels in the top stack frame agrees with the global times associated with the channels. This is guaranteed to be satisfied when the process backtracks to the initial state but might, as in the current example, be satisfied earlier. In this case, forward execution resumes with the stable region saved in the top stack frame:

$$\{\ell \mapsto (n_1, 2, n_2)\} \,⨾\, \langle n_1@3 : \Gamma, E_1[(\mathsf{stable}\ f_1)\ 100]\rangle$$

As illustrated in the example above, our semantic rules impose as few constraints as possible on the extent of, and the number of steps taken during backtracking, to serve as a general specification; our Scala implementation constrains the application of these rules to obtain an efficient implementation.

Formally, we have four rules. The rule H5 allows a process to asynchronously enter a backtracking state. Rule H6 allows a pair of communicating processes in the backtracking state (either because they encountered the backtracking command itself or asynchronously decided to backtrack using the rule above) to select any earlier time for their common channel. The label $\overline{\ell}@t'_c$ means that all communication events on channel ℓ at times later than t'_c are retracted. Notice that only the channel time is reduced –

this preserves our invariant that the virtual time of every process is greater than or equal to that of its channels. Rule H7 allows a process in the backtracking state to pop stack frames that were pushed after the required reset time of the channel. Finally, in rule H8, a process that is backtracking can return to forward action if all its channels are in a "consistent" state – that is, when all channels in its stored channel map in the top stack frame have timestamps matching what is found in the global channel map. It is only at this point that the virtual time of the process is updated.

4 Channel Protocol

The high-level semantics assumes that synchronous communication is realized atomically (e.g., rule H2). In an actual implementation, synchronization between sender and receiver requires a multi-phase handshaking protocol. It is not evident that such low-level protocols are robust if interleaved with backtracking actions. Because of the subtlety of this point, we formalize a low-level communication protocol with backtracking and prove it correct. In the next section, we will introduce a low-level semantics and use invariants of the protocol to prove that the high-level semantics can be faithfully implemented without assuming atomic synchronous communication.

4.1 Low-Level Communication

Instead of assuming that synchronous communication is atomic, we consider instead the realistic situation in which communication happens in two phases: (i) a sender requests a communication event, and (ii) after some unspecified time the receiver acknowledges the request.

In a distributed environment, where channel state changes made by the sender or receiver take time to propagate, the low-level communication messages introduce potential race conditions. We account for race conditions by verifying a model where communication is buffered. Thus, global channel state will be divided into two parts – one maintained by the sender and the other maintained by the receiver. A process may only write the state associated with its channel end, and may only read a delayed version of the state maintained by its communicating partner. This requires that channels carry two timestamps – one maintained by the sender and one maintained by the receiver. We think of the timestamp maintained by the receiver as the "true" channel time. In addition to independent timestamps, each end of the channel will also have a token bit and a "direction" flag. The token bits jointly determine which end of the channel may make the next "move," and the flag (loosely) determines the direction of communication, forward or backwards.

A crucial aspect exposed by the low-level communication protocol is the ability of a blocked sender or receiver engaged in a synchronization to signal its partner that it wishes to switch from forward to backwards communication. To accommodate such situations, the receiver state will also include an auxiliary Boolean variable `sync` that is set when the receiver agrees to complete a communication event and reset when it refuses a communication event. This variable is not visible to the sender; however, the sender will be able infer its value from the visible state even in the presence of potential races.

4.2 Protocol Types

We now introduce the formal model for the channel protocol using the Symbolic Analysis Laboratory (SAL) tools [20, 19, 4]. In the SAL language the key protocol types are defined as:

```
TIME : TYPE = NATURAL;
DIR  : TYPE = { B, I, F }; % backwards, idle, forward
```

As motivated above, each of the sender and receiver states are defined by four state variables:

```
OUTPUT s_b  : BOOLEAN   % sender token
OUTPUT s_t  : TIME      % sender time
OUTPUT s_d  : DIR       % sender direction
OUTPUT v    : NATURAL   % sender data

OUTPUT r_b   : BOOLEAN  % receiver token
OUTPUT r_t   : TIME     % receiver time
OUTPUT r_d   : DIR      % receiver direction
```

The state of a channel consists of the union of the sender and receiver states. In general, the right to act alternates between the sender and the receiver. The sender is permitted to initiate a communication event (forwards or backwards) when the two token bits are equal. The receiver is permitted to complete a communication event when the two token bits are unequal. Thus the sender (receiver) "holds" the token when these bits are equal (unequal). This alternating behavior is a characteristic of handshake protocols.

4.3 Model

In SAL, transition rules are simply predicates defining pre- and post-conditions; the next state of `s_b` is `s_b'`. Forward communication is initiated when the sender executes the guarded transition:

Trans 1.
```
(s_b = r_b) AND (r_d = F)  -->  s_b'  = NOT s_b; s_d'  = F; s_t'  IN { x : TIME | x > r_t} ;
v' IN { x : NATURAL | true}
```

Thus, the sender may initiate forward communication whenever it holds the "token" (`s_b = r_b`) and the receiver is accepting forward transactions (`r_d = F`). By executing the transition, the sender selects a new time (`s_t'`), relinquishes the token, indicates that it is executing a forward transaction (`s_d' = F`), and selects (arbitrary) data to transfer.

The receiver completes the handshake by executing the following transition in which it updates its clock (to a value at least that offered by the sender), and flips its token bit. This transition is only permitted when both the sender and the receiver wish to engage in forward communication:

Trans 2.
```
(s_b /= r_b) AND (s_d /= B) AND (r_d = F) -->  r_b' = s_b;
r_t' IN { x : TIME | x >= s_t};
```

A receiver may also refuse a forward transaction by indicating that it desires to engage only in backwards communication:

Trans 3.
```
(s_b /= r_b) AND (s_d = F) --> r_b' = s_b; r_d' = B;
```

Our protocol also supports backwards communication events. The sender may initiate a backwards event whenever it holds the token:

Trans 4.
```
(s_b = r_b) --> s_b' = NOT s_b; s_t' IN { x : TIME | x < r_t} ;  s_d' = B
```

In a manner analogous to forward communication, the receiver may complete the event by executing the following transition. One subtlety of this transition is that the receiver may also signal whether it is ready to resume forward communication (`r_d' = F`) or wishes to engage in subsequent backward events (`r_d'= B`). The latter occurs when the sender has offered a new time that is not sufficiently in the past to satisfy the needs of the receiver:

Trans 5.
```
(s_b /= r_b) AND (s_d = B) --> r_b' = s_b;
r_d' IN { x : DIR | x = B or x = F }; r_t' = s_t;
```

While the protocol presented supports both forward and backwards communication, the sender may be blocked waiting for a response from a receiver when it wishes to backtrack. The following transition allows the sender to *request* that a forward transaction be retracted:

Trans 6. `(s_b /= r_b) AND (s_d = F) --> s_d' = I`

The receiver may either accept the original offer to communicate (*Trans* 2) or allow the retraction:

Trans 7. `(s_b /= r_b) AND (s_d = I) -->  r_b' = s_b; r_d' IN {x : DIR | x =  r_d or x = B};`

Similarly a blocked receiver may signal the sender that it wishes to backtrack:

Trans 8. `(s_b = r_b) -->  r_d' = B`

4.4 Key Invariant

The main subtleties that we need to verify occur when a sender may attempt to retract a forward request while the receiver simultaneously acknowledges that request, or when a receiver may decide, after it has acknowledged a request, that it wishes to backtrack. In either case the later decision "overwrites" state that may or may not have been seen by the partner. For our semantic model, it is crucial that the sender be able to determine whether the synchronization event occurred or was successfully retracted. The key invariant of our SAL model uses a shadow variable to prove this property.

5 Low-Level Processes and Refinement Mapping

Our low-level model is derived from the high-level model by implementing those rules involving synchronization using finer-grained rules based upon the the protocol model. Necessarily, there are more transition rules associated with the low-level model. For example, the single high-level transition implementing forward communication requires three transitions (two internal and one external or visible) in the low-level model. High level transitions not involving communication (H1, H3, H4, and H7) are adopted in the low-level model with minimal changes to account for the differences in channel state. The high level transitions H5 and H8 are adopted with a few additional side conditions.

Along with our presentation of the low-level model (henceforth LP), we sketch a refinement mapping from LP to the high-level model (henceforth HP) – essentially a function that maps every state of LP to a state of HP and where every transition of LP maps to a transition (or sequence of transitions) of HP. [1]

5.1 Low-Level Synchronization

The low-level model is defined by its own set of transition rules (Figure 3) along with additional state information relating to the channel implementation. We begin by discussing in those transitions relating to forward communication. We use these transitions to illustrate how low-level events "map" to high-level events. The high-level semantics includes rule H2 (forward communication) that requires simultaneous changes in two processes and rule H6 that depends upon the state of two processes; none of the low-level rules affects more than one process and the only preconditions on any of our low-level rules are the state of a single process and the state of its channels. Furthermore, these rules modify only the process state and the portion of a channel state (send or receive) owned by the process.

We define the state of a channel as a tuple $(s:(n,t,b,d,v), r:(n,t,b,d))$ where s is the state of the sender and r is the state of the receiver; $s.n$ is the sender id and $r.n$ is the receiver id. As mentioned, the

(L1) $\Xi\{\ell \mapsto (s:(n,t_s,b,-,-), r:(n_r,t_r,b,F)\} ⨾ \langle n@t:\ \Gamma, E[\mathsf{send}\ \ell\ v]\rangle \xrightarrow{\varepsilon}$
$\Xi\{\ell \mapsto (s:(n,t'_s,\overline{b},F,v), r:(n_r,t_r,b,F)\} ⨾ \langle n@t:\ \Gamma, E[\underline{\mathsf{send}}\ \ell\ v]\rangle$
where $t'_s > t_r$

(L2) $\Xi\{\ell \mapsto (s:(n_s,t_s,b,d_s,v), r:(n,t_r,\overline{b},F)\} ⨾ \langle n@t:\ \Gamma, E[\mathsf{recv}\ \ell(x).e]\rangle \xrightarrow{\ell @ t[v]}$
$\Xi\{\ell \mapsto (s:(n_s,t_s,b,d_s,v), r:(n,t,b,F)\} ⨾ \langle n@t:\ \Gamma, E[e[v/x]]\rangle$
where $t > \max(t_s,t)$ and $d_s \in \{F,I\}$

(L3) $\Xi\{\ell \mapsto (s:(n,t_s,b,d_s,v), r:(n_r,t_r,b,d_r)\} ⨾ \langle n@t:\ \Gamma, E[\underline{\mathsf{send}}\ \ell\ v]\rangle \xrightarrow{\varepsilon}$
$\Xi\{\ell \mapsto (s:(n,t_s,b,d_s,v), r:(n_r,t_r,b,d_r)\} ⨾ \langle n@t_r:\ \Gamma, E[()]\rangle$
where $d_s \neq B$ and $t_s \leq t_r$

(L4) $\Xi\{\ell \mapsto (s:(n_s,t_s,b,F,v), r:(n,t_r,\overline{b},-)\} ⨾ \langle n@t:\ \Gamma, E[\mathsf{backtrack}\ v]\rangle \xrightarrow{\varepsilon}$
$\Xi\{\ell \mapsto (s:(n_s,t_s,b,F,v), r:(n,t_r,b,B)\} ⨾ \langle n@t:\ \Gamma, E[\mathsf{backtrack}\ v]\rangle$
where $t_r > 0$

(L5) $\Xi\{\ell \mapsto (s:(n,t_s,b,-,v), r:(n_r,t_r,b,d)\} ⨾ \langle n@t:\ \Gamma, E[\mathsf{backtrack}\ v]\rangle \xrightarrow{\varepsilon}$
$\Xi\{\ell \mapsto (s:(n,t'_s,\overline{b},B,v), r:(n_r,t_r,b,d)\} ⨾ \langle n@t:\ \Gamma, E[\mathsf{backtrack}\ v]\rangle$
where $t'_s < t_r$

(L6) $\Xi\{\ell \mapsto (s:(n_s,t_s,\overline{b},B,v), r:(n,t,b,-)\} ⨾ \langle n@t:\ \Gamma, E[\mathsf{backtrack}\ v]\rangle \xrightarrow{\overline{\ell} @ t_s}$
$\Xi\{\ell \mapsto (s:(n_s,t_s,\overline{b},B,v), r:(n,t_s,\overline{b},F)\} ⨾ \langle n@t:\ \Gamma, E[\mathsf{backtrack}\ v]\rangle$

(L7) $\Xi\{\ell \mapsto ((n_s,t_s,b,d_s,v),(n,t_r,b,F)\} ⨾ \langle n@t:\ \Gamma, E[\mathsf{backtrack}\ v]\rangle \xrightarrow{\varepsilon}$
$\Xi\{\ell \mapsto ((n_s,t_s,b,d_s,v),(n,t_r,b,B)\} ⨾ \langle n@t:\ \Gamma, E[\mathsf{backtrack}\ v]\rangle$
where $t_r > 0$

(L8) $\Xi\{\ell \mapsto (s:(n,t_s,\overline{b},F,v), r:(n_r,t_r,b,d_r)\} ⨾ \langle n@t:\ \Gamma, E[\underline{\mathsf{send}}\ \ell\ v]\rangle \xrightarrow{\varepsilon}$
$\Xi\{\ell \mapsto (s:(n,t_s,\overline{b},I,v), r:(n_r,t_r,b,d_r)\} ⨾ \langle n@t:\ \Gamma, E[\underline{\mathsf{send}}\ \ell\ v]\rangle$

(L9) $\Xi\{\ell \mapsto ((n_s,t_s,\overline{b},I,v),(n,t_r,b,d)\} ⨾ \langle n@t:\ \Gamma, E[e]\rangle \xrightarrow{\varepsilon}$
$\Xi\{\ell \mapsto ((n_s,t_s,\overline{b},I,v),(n,t_r,\overline{b},d')\} ⨾ \langle n@t:\ \Gamma, E[e]\rangle$
where $d' \in \{d,B\}$

(L10) $\Xi\{\ell \mapsto ((n,t_s,b,I,v),(n_r,t_r,b,d_r)\} ⨾ \langle n@t:\ \Gamma, E[\underline{\mathsf{send}}\ \ell\ v]\rangle \xrightarrow{\varepsilon}$
$\Xi\{\ell \mapsto ((n,t_s,b,I,v),(n_r,t_r,b,d_r)\} ⨾ \langle n@t:\ \Gamma, E[\mathsf{send}\ \ell\ v]\rangle$
where $t_r > t_s$

(H8 Condition) $(\Xi\{\ell \mapsto (s:(n_1,-,b,-), r:(-,t,b,-))\} \vee \Xi\{\ell \mapsto (s:(-,-,-,-), r:(n_1,t,-,F))\})$

Figure 3: Low-level rules

sender and receiver both maintain (non-negative) timestamps ($s.t$, $r.t$) and Boolean tokens ($s.b$, $r.b$). Each also maintains a direction flag ($s.d$, $r.d$) indicating "forward" or "backward" synchronization. The sender state includes a value $s.v$ to be transferred when communication occurs. These state elements correspond to those of the channel protocol. The refinement mapping from LP to HP drops this additional channel state information; although it does impact the mapping of the process expressions.

Forward communication (H2 in HP) is executed in three steps by the underlying channel protocol.

L1 In the first step, which corresponds to *Trans* 1 of the SAL model, the sender initiates the communication by marking its state as "in progress" with the new expression $\underline{\text{send}}\ \ell\ v$. This expression has no direct equivalent in HPand may only occur as a result of this rule.

   Recall that the sender has the "token" when the two channel token bits are equal ($s.b = r.b$), and initiates communication by inverting its token bit ($s.b$).

L2 In the second step, (*Trans* 2) the receiver "sees" that the sender has initiated communication, reads the data, updates its local virtual time, updates the channel's time, and flips its token bit to enable the sender to take the next and final step in the communication. (Note that the sender stays blocked until the receiver takes this step.) After taking this step the receiver can proceed with its execution:

   From L1 we can show that $t_s > t_r$. A required invariant for our model is that when the conditions of this rule are satisfied, the sender n_s is executing $\underline{\text{send}}\ \ell$.

L3 In the final step, the sender notes that its active communication event has been acknowledged by the receiver. It updates its local time and unblocks.

The introduction of new control states such as $\underline{\text{send}}\ \ell\ v$ necessarily complicates the creation of a refinement mapping, which maps this to either $\text{send}\ \ell\ v$ or $()$ depending upon the state of the channel. This follows naturally from the protocol in which a sender initiates a synchronization event, but the receiver completes it. Consider the following cases for mapping of $\underline{\text{send}}\ \ell\ v$. The first case corresponds to the state after transition L1 and the second to the state after transition L2. (Recall that ℓ is a channel, and $\ell.x.y$ correspond to fields of the channel state).

$$f(E[\underline{\text{send}}\ \ell\ v]) = E[\text{send}\ \ell\ v]\ \texttt{if}\ (\ell.s.b \neq \ell.r.b) \vee \ell.s.t > \ell.r.t$$
$$f(E[\underline{\text{send}}\ \ell\ v]) = E[()]\ \texttt{if}\ (\ell.s.b = \ell.r.b) \wedge \ell.s.t \leq \ell.r.t$$

Thus in the mapping, L1 is a "silent" (stuttering) transition and L2, which is only executed in parallel with a process executing $\underline{\text{send}}\ \ell\ v$, corresponds to high-level transition H2. Notice that the mapping of this control state depends upon the state of channel ℓ, although the mapping drops this additional state information from the channels. Finally, L3 maps to HP as a silent transition.

5.2 Backtracking Communication

Backwards communication is considerably more complex for several reasons. First, consider that a process wishing to backtrack may communicate with any of its peers – at the low level this may involve simultaneous communication along its various channels. A further complication has to do with the fact that low-level communication involves a handshake between the sender and receiver where the sender "commits" and then the receiver may acknowledge. If a sender has committed on a channel and detects, through its other channels, that a peer wishes to backtrack it must somehow retract the outstanding communication event.

As with forward communication, many of the transitions relating to backwards communication implement specific transitions in the channel model.

L5 A sender may initiate a backwards event (H6 in HP) if it is in the backtracking state and "holds" the token (*Trans* 4).

Notice that the sender's time did not change; the time will change when the sender transitions from backwards to forwards operation (see H8)

L6 A backtracking channel receiver may acknowledge the backwards transaction (*Trans* 5).

L7 A receiver that is backtracking may need to signal a sender that it wishes to backtrack (*Trans* 8).

Notice this simply raises a flag requesting a backtracking event. It is up to the sender to pay attention by requesting a backwards communication event (after entering the backtracking state if necessary).

L4 Finally, a receiver may reject a request for forward communication (this should only occur if the receiver is backtracking) (*Trans* 3). Notice that in this case $t_s > t_r$.

5.2.1 Retracting Forward Requests

As discussed in Sec. 4, we have special actions that allow a blocked sender or receiver to request backward communication.

L8 If the sender is blocked, it may ask the receiver to allow it to retract the communication request – this should only be executed where the sender has been requested to backtrack through some other channel (*Trans* 6).

L9 The receiver *may* allow the sender to retract the communication (*Trans* 7) or it may acknowledge the communication using the rule L2 above. We have a proof obligation to show that the n_s is executing <u>send</u> ℓ v whenever $\ell.s.d = I$. Furthermore, we have an obligation to show that $t_s > t_r$.

L10 Once a sender has been permitted to retract its request, it may return to the sending state from which it is free to respond to requests to backtrack. However, it is important to note that the receiver *may* have completed the request (L3). These two cases are covered by the invariant in our SAL model.

5.2.2 Initiating and Exiting Backtracking

There are two ways a process can begin backtracking – explicitly through a `backtrack` command in the program text or spontaneously. The low-level semantic rule for explicitly entering the backtracking state is the same as the high level rule (H5). However, because communication involves a series of protocol steps, we restrict spontaneous backtracking to the case where a process has no outstanding send request (i.e. it is not in the <u>send</u> state).[2] Note that the rule L8 allows a blocked sender to request retraction of an outstanding request and hence transition to a state where this side condition is satisfied.

Finally, we need a rule that allows a process to exit the backtracking state – H8 with constraints. The conditions on timestamps are the same, but send channels are required to be in a quiescent state. Since the receiver can only change the timestamp when it has the token (non-quiescent state), this predicate can safely be evaluated sequentially. For process n_1 and channel l this side condition is (H8 Condition) in the figure. Although checking this condition requires testing the state of all the process' channels, the channel protocol guarantees that if a channel satisfies the necessary condition then that property is

[2] Our Scala implementation further restricts spontaneous backtracking transitions to situations where a neighbor is backtracking; however, that does not alter the underlying model.

stable. Note that for send channels we require that the sender holds the token, and receive channels we require that the process has not requested further backtracking (this may occur in the model, with its non-determinism, but does not occur in our implementation).

6 Conclusion

We have introduced a CSP based language supporting reversible distributed computing along with two semantic models – a high-level model in which synchronous events are modeled by transitions that affect two processes simultaneously, and a low-level model in which transitions affect a single process. These two models are related by a verified communication protocol which is the basis for the finer grained transitions of the low-level model. We outlined a refinement mapping that we developed proving that the low-level model implements the high-level model. This proof required the invariants of the protocol that were verified with the SAL model checker. We have also proved that the high-level model obeys sensible causal ordering properties even in the face of backtracking.

While our Scala language implementation is somewhat richer than the simple models presented here (e.g., it supports communication choice, and dynamic process and channel creation); at its core it is implemented exactly as indicated by our low-level model. Channels are implemented via message passing where the messages carry the channel state of our protocol. Processes are implemented as Java threads. Processes learn that their peers wish to backtrack by examining the (local) state of their channels. Stable sections consist of: saving the channel timestamps on the context stack, executing the stable code in a try/catch block, and popping the stack; backtracking is implemented by throwing an exception. The implementation required approximately 1200 lines of code.[3]

This paper provides clear evidence that implementing reversible communicating processes in a distributed manner is both feasible and, from the perspective of communication overhead, relatively efficient. We note that our high-level model is somewhat unsatisfying because it exposes the programmer to the mechanics of backtracking. In our current model, even when a process has decided that it wishes to backtrack, its peers may continue forward execution for a period during which they may learn from their peers. If we were to restrict our attention to traditional roll-back recovery, where nothing is "learned" from unsuccessful forward execution, this could easily be abstracted. We continue to work towards a "compromise" between traditional rollback and the unrestricted model we have presented.

Acknowledgment. This material is based upon work supported by the National Science Foundation under grant numbers 1116725 and 1217454 and while the first author was an NSF program officer. Any opinion, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] Martín Abadi & Leslie Lamport (1991): *The existence of refinement mappings*. *TCS* 82(2), pp. 253 – 284, doi:10.1016/0304-3975(91)90224-P.

[2] Vincent Danos & Jean Krivine (2005): *Transactions in RCCS*. In: *CONCUR 2005, LNCS* 3653, Springer, pp. 398–412, doi:10.1007/11539452_31.

[3] Download: http://cs.indiana.edu/∼geobrown/places-code.tar.gz.

[3] Vincent Danos, Jean Krivine & Fabien Tarissan (2007): *Self-assembling Trees*. *ENTCS* 175(1), pp. 19 – 32, doi:10.1016/j.entcs.2006.11.017.

[4] Bruno Dutertre & Maria Sorea (2004): *Timed Systems in SAL*. Technical Report SRI-SDL-04-03, SRI International.

[5] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang & David B. Johnson (2002): *A Survey of Rollback-recovery Protocols in Message-passing Systems*. *ACM Comput. Surv.* 34(3), pp. 375–408, doi:10.1145/568522.568525.

[6] Colin J. Fidge (1988): *Timestamps in message-passing systems that preserve the partial ordering*. In: *Proc. of the 11th Australian Computer Science Conference (ACSC'88)*, pp. 56–66.

[7] Rachid Guerraoui (2010): *Foundations of Speculative Distributed Computing*. In: *Distributed Computing*, *LNCS* 6343, Springer, pp. 204–205, doi:10.1007/978-3-642-15763-9_21.

[8] M.V. Hermenegildo & K.J. Greene (1991): *The &-Prolog system: Exploiting independent and-parallelism*. *New Generation Computing* 9(3-4), pp. 233–256, doi:10.1007/BF03037164.

[9] Ralf Hinze (2000): *Deriving Backtracking Monad Transformers*. In: *ICFP '00*, ACM, pp. 186–197, doi:10.1145/351240.351258.

[10] Tony Hoare (2010): *Compensable Transactions*. In: *Advanced Lectures on Software Engineering*, *LNCS* 6029, Springer, pp. 21–40, doi:10.1007/978-3-642-13010-6_2.

[11] D. Jefferson, B. Beckman, F. Wieland, L. Blume & M. Diloreto (1987): *Time Warp Operating System*. In: *SOSP '87*, ACM, pp. 77–93, doi:10.1145/41457.37508.

[12] David R. Jefferson (1985): *Virtual Time*. *ACM Trans. Program. Lang. Syst.* 7(3), pp. 404–425, doi:10.1145/3916.3988.

[13] Ivan Lanese, ClaudioAntares Mezzina, Alan Schmitt & Jean-Bernard Stefani (2011): *Controlling Reversibility in Higher-Order Pi*. In: *CONCUR 2011*, *LNCS* 6901, Springer, pp. 297–311, doi:10.1007/978-3-642-23217-6_20.

[14] Ivan Lanese, ClaudioAntares Mezzina & Jean-Bernard Stefani (2010): *Reversing Higher-Order Pi*. In: *CONCUR 2010*, *LNCS* 6269, Springer, pp. 478–493, doi:10.1007/978-3-642-15375-4_33.

[15] Mohsen Lesani & Jens Palsberg (2011): *Communicating Memory Transactions*. In: *PPoPP*, ACM, pp. 157–168, doi:10.1145/1941553.1941577.

[16] Jing Li, Huibiao Zhu & Jifeng He (2008): *Specifying and Verifying Web Transactions*. In: *FORTE 2008*, *LNCS* 5048, Springer, pp. 149–168, doi:10.1007/978-3-540-68855-6_10.

[17] Michael Lienhardt, Ivan Lanese, ClaudioAntares Mezzina & Jean-Bernard Stefani (2012): *A Reversible Abstract Machine and Its Space Overhead*. In: *Formal Techniques for Distributed Systems*, *LNCS* 7273, Springer, pp. 1–17, doi:10.1007/978-3-642-30793-5_1.

[18] Friedemann Mattern (1988): *Virtual Time and Global States of Distributed Systems*. In: *Parallel and Distributed Algorithms*, North-Holland, pp. 215–226.

[19] Leonardo de Moura (2004): *SAL: Tutorial*. Technical Report, SRI International.

[20] Leonardo de Moura, Sam Owre & N. Shankar (2002): *The SAL Language Manual*. Technical Report, SRI International.

[21] Iain Phillips & Irek Ulidowski (2007): *Reversibility and Models for Concurrency*. *ENTCS* 192(1), pp. 93 – 108, doi:10.1016/j.entcs.2007.08.018.

[22] John H. Reppy (1999): *Concurrent Programming in ML*. Cambridge University Press.

[23] Edsko de Vries, Vasileios Koutavas & Matthew Hennessy (2010): *Communicating Transactions*. In: *CONCUR 2010*, *LNCS* 6269, Springer, pp. 569–583, doi:10.1007/978-3-642-15375-4_39.

[24] Lukasz Ziarek & Suresh Jagannathan (2010): *Lightweight Checkpointing for Concurrent Ml*. *J. Funct. Program.* 20(2), pp. 137–173, doi:10.1017/S0956796810000067.