Reversible Communicating Processes

Geoffrey Brown and Amr Sabry

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

Abstract

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 programming 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 [10]. 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 [12]. 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 [15, 14]), in models of concurrent languages (e.g.[2, 3, 16, 17, 22]) and in some restricted parallel shared-memory environments (e.g. [6, 11, 18, 23, 20]), 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 [19, 13]. Communicating message transactions [6, 18] 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 [8]. 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 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, which is outlined in the paper.

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 fragment of) a "low-level" model that utilizes the channel protocol to implement the high-level model, and Sec. 6 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

We illustrate the main ideas of our language with an example written in our Scala realization. A programmer wishing to use our distributed reversible extensions 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 following excerpt 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:

```
 1   class p1 (c: SndPort, name: String)
         extends CspProc(name) {
 3     override def uCode = {
         println("p1: start")
 5       var count = 2
         stable {
 7         println("p1: snd " + count)
           send(c,count)
 9         stable {
             println("p1: snd " + count)
11           send(c,count)
             count = count - 1
13           if (count > 0) {
               println ("p1: backtrack")
15             backtrack }
           }}
17     }}
```

```
 1   class p2 (c: RcvPort, name: String)
         extends CspProc(name) {
 3     override def uCode = {
         println("p2: start")
 5       stable {
           var x = receive(c)
 7         println("p2: recv " + x)
           var y = receive(c)
 9         println("p2: recv " + y)
         }
11     }}
```

Output:

```
p1: start
p2: 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
```

In the code, 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. Process `p1` starts its execution by sending a first message to `p2`, entering a nested stable region, and sending another message to `p2`. Meanwhile process `p2` also starts a stable region in which it receives the two messages. At this point in the execution, process `p1` decides at line 15 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 8) was invalidated, it backtracks to the start of its stable region. This jump invalidates the first communication action (line 6) 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. The trace in the output shows a possible interleaving of the execution – it is important to remember that all processes have an implicit stable region that includes their full code body; in this case `p2` is forced to backtrack to the beginning of its code. The crucial point is that after the backtracking, 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 [4, 5, 7]; 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 [9, 21]. Our approach differs in utilizing synchronous communication and also by providing a fully distributed rollback protocol.

3.1 User-Level Syntax

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

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

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 receives x on channel ℓ. Both our Scala implementation and full model support input "choice"; we have omitted the details 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 the semantics, 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 and are collected below:

(*process name*)	n		
(*time stamp*)	t		
(*values*)	v	$::=$	$c \mid x \mid \lambda x.e \mid \textsf{stable}\ (\lambda x.e)$
(*expressions*)	e	$::=$	$\ldots \mid \underline{\textsf{stable}}\ e$
(*evaluation contexts*)	E	$::=$	$\square \mid E\ e \mid v\ E \mid \textsf{send}\ \ell\ E \mid \textsf{stable}\ E \mid \underline{\textsf{stable}}\ E \mid \textsf{backtrack}\ E$
(*channel map*)	Ξ	$=$	$\ell \mapsto (n, t, n)$
(*stacks*)	Γ	$=$	$\bullet \mid \Gamma, (E, t, \Xi)$
(*processes*)	p	$::=$	$p_1 \parallel p_2 \quad \mid \quad \langle n@t :\ \Gamma, e\rangle$

Expressions are extended with $\underline{\textsf{stable}}\ e$ 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 a global map 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), a time stamp, and a local channel map describing the state of the communication channels at the time of the checkpoint. 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. We assume that in the initial system state, all channels have time 0 and every process is of the form $\langle n@0 :\ \bullet, \textsf{stable}\ (\lambda\_.e)\ ()\rangle$; i.e. process n is entering a stable region containing the expression e with an empty context stack at time 0.

We now informally describe the key semantic specifications that the above structures must satisfy and illustrate some of the key points using examples. Aiming for clarity, the presentation is simplified in several inessential aspects that are formalized in the remaining sections. Any computation step that does not involve communication, stable regions, or backtracking is considered a local computation step. None of the above structures need to be consulted or updated during such local computation steps. In particular, the virtual time stamp is not incremented and local computation steps proceed at "full native speed." In order to establish notation for the remaining examples, here is a simple computation step:

$$\{\ell_1 \mapsto (p_1, 2, p_2)\} \fatsemi \langle p_1@5 :\ \bullet, 1+2\rangle \rightarrow \{\ell_1 \mapsto (p_1, 2, p_2)\} \fatsemi \langle p_1@5 :\ \bullet, 3\rangle$$

In this example, a process p_1 with local virtual time 5 and making forward progress encounters the computation $1+2$. The process's context stack is currently empty ($\bullet$).[1] We also assume the existence of a channel ℓ_1 which is associated with time 2 and connects p_1 to p_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.

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 establish the following invariant. At the end of handshake, the virtual times of the sending process, the receiving process, and the used channel are all equal, and that this new virtual time is larger than any of the prior times for these structures. Here is a small example illustrating this communication handshake:

$$\begin{array}{rl} & \{\ell_1 \mapsto (p_1, 3, p_2)\} \fatsemi \langle p_1@5 :\ \bullet, \textsf{send}\ \ell_1\ 10\rangle \parallel \langle p_2@4 :\ \bullet, \textsf{recv}\ \ell_1(x).x+1\rangle \\ \rightarrow & \{\ell_1 \mapsto (p_1, 6, p_2)\} \fatsemi \langle p_1@6 :\ \bullet, ()\rangle \parallel \langle p_2@6 :\ \bullet, 10+1\rangle \end{array}$$

Initially, we have two processes willing to communicate on channel ℓ_1. Process p_1 is sending the value 10 and process p_2 is willing to receive an x on channel ℓ_1 and proceed with $x+1$. After the reduction, the value 10 is exchanged and each process proceeds to the next step. The important invariant that has been

[1] Actually, each process starts with an implicit stable region which is omitted for clarity.

established is that the virtual times of the two processes as well as the virtual time of the channel ℓ_1 have all been synchronized to time 6 which is greater than any of the previous times.

When a process executing in the forward direction encounters a new stable region, it increments its virtual time and pushes a new context containing a continuation (or an exception handler) and the current virtual times of all its current communication ports. If the execution of this stable region ends "normally," the context is popped and forward execution continues. As an example, consider:

$$\begin{array}{rl} & \{\ell_1 \mapsto (p_1, 2, p_2)\} \fatsemi \langle p_1@5 : \bullet, 7 + (\mathsf{stable}\ f)\ v\rangle \\ \rightarrow & \{\ell_1 \mapsto (p_1, 2, p_2)\} \fatsemi \langle p_1@6 : \bullet, (7 + (\mathsf{stable}\ f)\ \square, 5, \{\ell_1 \mapsto (p_1, 2, p_2)\}), 7 + \underline{\mathsf{stable}}\ (f\ v)\rangle \end{array}$$

Process p_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 p_1 pushes $(7 + (\mathsf{stable}\ f)\ \square, 5, \{\ell_1 \mapsto (p_1, 2, p_2)\})$ 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 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_1 and then returning the value 100, then the evaluation progresses as follows:

$$\begin{array}{rl} & \{\ell_1 \mapsto (p_1, 8, p_2)\} \fatsemi \langle p_1@8 : \bullet, (7 + (\mathsf{stable}\ f)\ \square, 5, \{\ell_1 \mapsto (p_1, 2, p_2)\}), 7 + \underline{\mathsf{stable}}\ 100\rangle \\ \rightarrow & \langle p_1@8 : \bullet, 7 + 100\rangle \end{array}$$

The context stack is popped and execution continues in the forward direction.

The more challenging situation occurs when the execution of $f\ v$ encounters a backtracking command (whether directly initiated by the process itself or indirectly initiated via a communicating partner). This case will be described in detail in the next section.

3.3 Forward Semantics

In the remainder of this section we define the high-level semantics through a set of transition rules on semantic configurations where a semantic configuration $\Xi \fatsemi p_1 \parallel p_2 \ldots$ consists of the global channel map Ξ and a number of processes.

Internal evaluation by a single process is captured with fairly conventional rules. We present the rule for application of λ-expressions; the rules for applying primitive operations are similar:

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

In the rule, 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 $e[v/x]$.[2]

The rule for forward communication is:

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

Where $t > \max(t_1, t_2)$. 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. Two processes communicate on a shared channel when one is prepared to send and the other is prepared to receive. The act of communication causes data to be transferred

[2]For readability, the color version of the paper highlights the components of the configuration that are modified by each rule.

from the sender to receiver and a new (later) virtual time to be assigned to each of the processes and the channel. An important model invariant is that $t_1 \geq t_c \wedge t_2 \geq t_c$. Notice further that this transition produces a visible event $\ell @ t[v]$ signifying that value v was transferred on channel ℓ at time t.

Finally there are rules corresponding to entering and exiting stable regions. Entry into a stable region causes a new context to be pushed onto the stack. Notice that a value is passed into the stable region being executed. The syntax $\underline{\mathsf{stable}}\ e$ means that e is being evaluated within a stable region:

$$\begin{array}{l} \Xi \,⨟\, \langle n@t:\ \Gamma, E[(\mathsf{stable}\ (\lambda x.e))\ v]\rangle \\ \xrightarrow{\epsilon} \\ \Xi \,⨟\, \langle n@t':\ (\Gamma, (E[(\mathsf{stable}\ (\lambda x.e))\ \square], t, \Xi_n)), E[\underline{\mathsf{stable}}\ e[v/x]]\rangle \end{array} \tag{H3}$$

Where $t' > t$ and Ξ_n is the subset of Ξ referring to the channels of n. The saved context $E[(\mathsf{stable}\ (\lambda x.e))\ \square]$ on the stack will be reinstated in case of a backtracking action. As the rules in the next section will show, 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 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 $()$. A well-typed program should have the function argument to stable ($\lambda x.e$ in the rule above) be prepared to handle either situation.

When a stable region is completed, the stored context is popped:

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

3.4 Backtracking Semantics

The novelty of our language is in its support of backtracking. Backtracking is initiated by a process that encounters a $\mathsf{backtrack}$ expression. A process executing a $\mathsf{backtrack}$ event can synchronize with other processes through "backwards communication events." Any process engaging in a backwards communication event is forced into the backtracking state:

$$\begin{array}{l} \Xi\{\ell \mapsto (c_1, t_c, c_2)\} \,⨟\, \langle n_1@t_1:\ \Gamma_1, e_1\rangle \parallel \langle n_2@t_2:\ \Gamma_2, E_2[\mathsf{backtrack}\ v]\rangle \\ \xrightarrow{\overline{\ell}@t} \\ \Xi\{\ell \mapsto (c_1, t, c_2)\} \,⨟\, \langle n_1@t_1:\ \Gamma_1, \mathsf{backtrack}\ ()\rangle \parallel \langle n_2@t_2:\ \Gamma_2, E_2[\mathsf{backtrack}\ v]\rangle \end{array} \tag{H5}$$

Where $0 \leq t < t_c$ and $\{c_1, c_2\} = \{n_1, n_2\}$. (Case where n_1 is already backtracking omitted).

A backwards communication event occurs between two processes sharing a channel where at least one of the processes is in the backtracking state. The label $\overline{\ell}@t$ means that all communication events on channel ℓ at times later than t 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. The virtual time of a backtracking process is only updated when it returns to forward execution.

While our semantic rules impose as few constraints as possible, our Scala implementation demonstrates that it is possible to constrain the application of these rules to obtain an efficient implementation.

A backwards communication event may bring a process to a state in which the top context on its stack is "later" than required by one of its channels. In this case, repeated backtracking is required – in effect popping its context stack.

$$\begin{array}{l} \Xi\{\ell \mapsto (-, t, -)\} \,⨟\, \langle n_1@t_1:\ \Gamma_1, (E_1, t_1', \Xi_1\{\ell \mapsto (-, t', -)\}), E[\mathsf{backtrack}\ v]\rangle \\ \rightarrow \\ \Xi \,⨟\, \langle n_1@t_1:\ \Gamma_1, E[\mathsf{backtrack}\ v]\rangle \end{array} \tag{H6}$$

Where $t < t'$. Because of our initial conditions, this rule is always possible if $0 < t'$ – in effect we guarantee there is a such a stored context.

Finally, 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 have time-stamps matching what is found in the global channel map.

$$\Xi \fatsemi \langle n_1@t_1 : \; \Gamma_1, (E_1, t_1', \Xi_1), E[\textsf{backtrack } v] \rangle \rightarrow \Xi \fatsemi \langle n_1@t_1' : \; \Gamma_1, E_1[v] \rangle \tag{H7}$$

Where $\Xi_1\{\ell \mapsto (-, t, -)\} \Rightarrow \Xi\{\ell \mapsto (-, t, -)\}$. Again, our initial conditions guarantee that such a state is possible.

4 Channel Protocol

The principal difference between the low-level (Section 5) and high-level semantics is that the low-level communication is implemented using a multi-phase handshaking protocol where the sender requests a communication event that the receiver subsequently acknowledges. The state corresponding to a channel is 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, but may read (a delayed version of) that maintained by its communicating partner. Necessarily, the channel implementation has two time-stamps – one maintained by the sender and one maintained by the receiver. It is convenient to think of the time-stamp maintained by the receiver as the "true" channel time. In addition to independent time-stamps, each end of the channel has 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.

We developed a formal model for the channel protocol using the Symbolic Analysis Laboratory (SAL) tools [5, 4, 7] and present the protocol using the SAL syntax. The invariants verified with the SAL model are crucial to the refinement proofs that link our two semantic models. For example, the SAL invariants show that if the processes using a channel obey the protocol, the various timestamps will obey the ordering assertions our semantics depend upon.

In the SAL language the key protocol types are defined as:

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

The sender state is 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
```

Similarly for the receiver state:

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

The state of the 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.

An important wrinkle in our protocol is the ability of blocked sender or receiver to signal its partner that it wishes to switch from forward to backwards communication. The receiver state also includes an auxiliary Boolean variable sync used to support the proof. The receiver sets sync when it completes a communication event and resets it when it refuses a communication event. This variable is not visible to the sender; however, our automated proof demonstrates that the sender can infer its value from the visible state even in the presence of potential races.

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_t'  IN { x : TIME | x > r_t} ;
                                s_d'  = F;
                                v' IN { x : NATURAL | true}
```

In the SAL language, transition rules are simply predicates defining pre- and post-conditions; the next state of `s_b` is `s_b'`. 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
```

In a distributed environment, where channel state changes made by the sender or receiver take time to propagate, these two rules introduce potential race conditions. Our SAL model accounts for race conditions by verifying a model where communication is buffered. Thus, the invariants proved using the SAL tools are valid in a distributed environment. Our SAL model includes a "shadow variable," `sync`, that the receiver sets whenever it accepts a communication event and clears whenever it rejects one. Whenever the sender holds the token, `sync` is true exactly when `r_t >= s_t`. Thus, the sender can determine which of the racing events occurred.

Consider that the sender may attempt to retract a forward request while the receiver simultaneously acknowledges that request. Similarly, the 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. Indeed we prove a key invariant:

```
(s_b = r_b) => ((s_t <= r_t)) = sync)
```

Thus, when the sender holds the token, it can determine whether its last communication request was completed.

5 Low-Level Processes

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 H6) are adopted in the low-level model with minimal changes to account for the differences in channel state. Due to space limitations, we consider only those transitions relating to forward communication. We use these transitions to illustrate how low-level events "map" to high-level events.

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 sender and receiver both maintain (non-negative) time-stamps ($s.t$, $r.t$) 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 the those of the channel protocol.

Where the high-level semantics included rules H2 and H5 that require simultaneous changes in two processes, none of the low-level rules affects more than one process. Indeed, 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.

Forward communication (H2 in the High-level model) is executed in three steps by the underlying channel protocol.

1. In the first step, which corresponds to **Trans 1** of the SAL model, the sender initiates the communication. The sender marks its state as "in progress" with the new expression $\underline{\textsf{send}}\ \ell\ v$, which has no direct equivalent in the high-level model and which may only occur as a result of this transition rule.

$$\begin{array}{l}\Xi\{\ell \mapsto (s : (n, t_s, b, -, -), r : (n_r, t_r, b, F)\} \fatsemi \langle n@t :\ \Gamma, E[\textsf{send}\ \ell\ v]\rangle \\ \xrightarrow{\epsilon} \\ \Xi\{\ell \mapsto (s : (n, t'_s, \bar{b}, F, v), r : (n_r, t_r, b, F)\} \fatsemi \langle n@t :\ \Gamma, E[\underline{\textsf{send}}\ \ell\ v]\rangle \end{array} \qquad \text{(L1)}$$

Where $t'_s > t_r$. 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 ($s.b$).

2. 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.

$$\begin{array}{l}\Xi\{\ell \mapsto (s : (n_s, t_s, b, d_s, v), r : (n, t_r, \bar{b}, F)\} \fatsemi \langle n@t :\ \Gamma, E[\mathsf{recv}\ \ell(x).e]\rangle \\ \xrightarrow{\ell@t[v]} \\ \Xi\{\ell \mapsto (s : (n_s, t_s, b, d_s, v), r : (n, t, b, F)\} \fatsemi \langle n@t :\ \Gamma, E[e[v/x]]\rangle\end{array} \tag{L2}$$

Where $t > \max(t_s, t)$ and $d_s \in \{F, I\}$. 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 $\underline{\mathsf{send}}\ \ell$.

3. 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.

$$\begin{array}{l}\Xi\{\ell \mapsto (s : (n, t_s, b, d_s, v), r : (n_r, t_r, b, d_r)\} \fatsemi \langle n@t :\ \Gamma, E[\underline{\mathsf{send}}\ \ell\ v]\rangle \\ \xrightarrow{\epsilon} \\ \Xi\{\ell \mapsto (s : (n, t_s, b, d_s, v), r : (n_r, t_r, b, d_r)\} \fatsemi \langle n@t_r :\ \Gamma, E[()]\rangle\end{array} \tag{L3}$$

Where $d_s \neq B$ and $t_s \leq t_r$.

6 Refinement Mapping Proof Outline

We have (partially) presented two semantic models and and a channel protocol. We can use the high-level model to inform programmers about expected program behavior, the low-level model to guide implementers, and the channel protocol to help prove invariant properties about the low-level model. In this section we outline the refinement mapping that we used to prove that the low level processes (henceforth LP); i.e., a function that maps every state of LP to a state of HP and where every transition of LP maps to a transition of HP.

Following Abadi and Lamport [1], a specification S is defined by (Σ, F, N) where Σ is a state space, F is the set of initial states, and N is the next-state relation. To prove that $S_1 = (\Sigma_1, F_1, N_1)$ implements $S_2 = (\Sigma_1, F_2, N_2)$ we need to define a mapping $f : \Sigma_1 \to \Sigma_2$ such that:

R2. For all $f(F_1) \subseteq F_2$ (f takes initial states into initial states.)

R3. If $\langle s, t\rangle \in N_1$ then $\langle f(s), f(t)\rangle \in N_2$ or $f(s) = f(t)$. (A state transition allowed by N_1 is mapped into a [possibly stuttering] transition allowed by N_2.)

Note that we have omitted the portions of the Abadi and Lamport definition that handle supplemental properties along with rules R1 and R4 which deal with externally visible state and supplemental properties (respectively).

The states of HP are defined by parallel composition of a finite set of processes as defined by syntax of Section 3.2. The initial states of HP are those where every context stack is empty, every process is of the form $\langle n@0 :\ \bullet, \mathsf{stable}\ (\lambda\_.e)\ ()\rangle$, and every channel is at time 0. The initial states of LP are exactly the initial states of HP but where each channel state has two additional token bits, both false, two clocks, both 0, and two direction flags, both F.

We can quickly dispense with mapping rule R2 by stating that our mapping function converts channel maps of LP to those of HP by dropping the additional state information, and maps the syntax to LP to that of HP except for two process forms, neither of which is permitted in the initial state:

$$\langle n@t :\ \Gamma, \underline{\mathsf{send}}\ \ell\ e\rangle$$
$$\langle n@t :\ \Gamma, \underline{\mathsf{backtrack}}\ \ell\ e\rangle$$

$\underline{\textsf{send}}\ \ell$ was discussed in Section 5 where it was used to capture the intermediate state of a forward communication transaction; $\underline{\textsf{backtrack}}\ \ell$ serves as an intermediate state for the backtracking rules that we have omitted due to space limitations.

As an example of mapping transition rules, consider the following cases for mapping of $\underline{\textsf{send}}\ \ell\ v$. 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).

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

Thus in the mapping, L1 is a "silent" transition and L2, which is only executed in parallel with a process executing $\underline{\textsf{send}}\ \ell\ v$, corresponds to high-level transition H2.

Validating the (complete) refinement mapping requires proving that every LP transition maps to a HP transition.

7 Discussion

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. As we noted, our "high-level" model is 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 and Leslie Lamport. The existence of refinement mappings. *Theor. Comput. Sci.*, 82(2):253–284, May 1991.

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

[2] V. Danos and J. Krivine. *Transactions in RCCS*, pages 398–412. Springer-Verlag, London, UK, 2005.

[3] V. Danos, J. Krivine, and F. Tarissan. Self-assembling trees. *Electron. Notes Theor. Comput. Sci.*, 175:19–32, May 2007.

[4] Leonardo de Moura. SAL: tutorial. Technical report, SRI International, 2004.

[5] Leonardo de Moura, Sam Owre, and N. Shankar. The SAL language manual. Technical report, SRI International, 2002.

[6] Edsko de Vries, Vasileios Koutavas, and Matthew Hennessy. Communicating transactions. In *Proceedings of the 21st international conference on Concurrency theory*, CONCUR'10, pages 569–583. Springer-Verlag, 2010.

[7] Bruno Dutertre and Maria Sorea. Timed systems in SAL. Technical Report SRI-SDL-04-03, SRI International, 2004.

[8] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, September 2002.

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

[10] Rachid Guerraoui. Foundations of speculative distributed computing. In Nancy Lynch and Alexander Shvartsman, editors, *Distributed Computing*, volume 6343 of *Lecture Notes in Computer Science*, pages 204–205. Springer-Verlag, 2010.

[11] M. Hermenegildo and K. Greene. The &-prolog system: Exploiting independent and-parallelism. *New Generation Computing*, 9:233–256, 1991.

[12] Ralf Hinze. Deriving backtracking monad transformers. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, ICFP '00, pages 186–197. ACM, 2000.

[13] Tony Hoare. Compensable transactions. In Peter Mller, editor, *Advanced Lectures on Software Engineering*, volume 6029 of *Lecture Notes in Computer Science*, pages 21–40. Springer Berlin Heidelberg, 2010.

[14] D. Jefferson, B. Beckman, F. Wieland, L. Blume, and M. Diloreto. Time warp operating system. In *Proceedings of the eleventh ACM Symposium on Operating systems principles*, SOSP '87, pages 77–93. ACM, 1987.

[15] David R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7:404–425, 1985.

[16] Ivan Lanese, Claudio Antares Mezzina, Alan Schmitt, and Jean-Bernard Stefani. Controlling reversibility in higher-order Pi. In *Proceedings of the 22nd international conference on Concurrency theory*, CONCUR'11, pages 297–311. Springer-Verlag, 2011.

[17] Ivan Lanese, ClaudioAntares Mezzina, and Jean-Bernard Stefani. Reversing higher-order Pi. In Paul Gastin and Franois Laroussinie, editors, *CONCUR 2010 - Concurrency Theory*, volume 6269 of *Lecture Notes in Computer Science*, pages 478–493. Springer Berlin Heidelberg, 2010.

[18] Mohsen Lesani and Jens Palsberg. Communicating memory transactions. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 157–168. ACM, 2011.

[19] Jing Li, Huibiao Zhu, and Jifeng He. Specifying and verifying web transactions. In Kenji Suzuki, Teruo Higashino, Keiichi Yasumoto, and Khaled El-Fakih, editors, *Formal Techniques for Networked and Distributed Systems FORTE 2008*, volume 5048 of *Lecture Notes in Computer Science*, pages 149–168. Springer Berlin Heidelberg, 2008.

[20] Michael Lienhardt, Ivan Lanese, Claudio Antares Mezzina, and Jean-Bernard Stefani. A reversible abstract machine and its space overhead. In Holger Giese and Grigore Rosu, editors, *Formal Techniques for Distributed Systems*, volume 7273 of *Lecture Notes in Computer Science*, pages 1–17. Springer-Verlag, 2012.

[21] Friedemann Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms*, pages 215–226. North-Holland, 1988.

[22] Iain Phillips and Irek Ulidowski. Reversibility and models for concurrency. *Electron. Notes Theor. Comput. Sci.*, 192:93–108, October 2007.

[23] John H. Reppy. *Concurrent programming in ML.* Cambridge University Press, New York, NY, USA, 1999.

[24] Lukasz Ziarek and Suresh Jagannathan. Lightweight checkpointing for concurrent ML. *J. Funct. Program.*, 20(2):137–173, March 2010.