Verifying a C implementation of Derecho's coordination mechanism using VST and Coq

Ramana Nagasamudram[1][0000−0003−2779−2071], Lennart Beringer[2][0000−0002−1570−3492], Ken Birman[3][0000−0003−2400−149X], Mae Milano[2][0000−0003−3126−7771], and David A. Naumann[1][0000−0002−7634−6150]

[1] Stevens Institute of Technology
[2] Princeton University
[3] Cornell University

Abstract. Derecho is a C++ framework for distributed programming leveraging high performance communication primitives such as RDMA. At its core is the shared state table (SST), a replicated data structure that supports efficient protocols for consensus and group membership. Our aim is to formalize the reasoning principles articulated by the designers, which focus on knowledge and monotonicity, as basis for highly assured high performance distributed applications. To this end we develop a high level model that exposes the SST principles in an application-friendly way. We use the model to specify and verify a re-implementation in C of the SST API. We validate the specifications by verifying simple applications that embody key parts of the Derecho protocols. The development is carried out using VST and Coq. This lays groundwork for verification of the full Derecho protocols and applications built on them.

1 Introduction

Distributed systems occur ubiquitously in modern computational infrastructures, ranging from sensor networks or communicating embedded systems in the automotive domain to the datacenter and the global communication infrastructure. As application domains differ in their requirements concerning the performance and the applicable notion of consensus, numerous protocols have been developed.

In the domain of datacenter networks, latency and throughput requirements have recently led to the emergence of protocols that exploit monotonicity [14] to reduce communication overhead while facilitating strong notions of consensus [17]. At the same time, the reliance of everyday life on cloud computing necessitates that the expected efficiency and functionality guarantees of these protocols ought to be substantiated by verified implementations. But research that connects formal analyses of protocol-level properties [45,2,36] to implementation-level verification frameworks is only emerging now [13,16,38,40]. Ultimately, assurance of such *comprehensively verified* systems should encompass implementation correctness (down to instructions and hardware), protocol properties, and application-level guarantees, in a provably gap-free manner, with machine-checkable proofs.

To explore this challenge in a concrete setting, this paper reports on a comprehensive verification effort for the key data structure of Derecho [17], a distributed system platform implemented in C++ that combines protocol efficiency with low-latency communication, as facilitated through CPU-bypass technologies such as RDMA (remote direct memory access) [32]. Our approach is to:

– isolate the data structure, *shared state table* (SST), as a separate code unit in C, with a simplified API that supports the key functionality but can potentially be retrofitted into Derecho;
– equip the API with an expressive specification that connects functional code correctness to an abstract model of local data structure operations and global synchronization events but does not unduly constrain concurrency (Sec. 3);
– formulate the designer's intuitions concerning monotonicity and knowledge-based consensus as abstract trace (hyper-)properties over the operations and events (Sec. 3);
– confirm applicability of the resulting programming model by verifying two applications, exploiting only the API-level specifications and the trace properties to establish application-level correctness guarantees (Sec. 4 and 6);
– verify the isolated SST implementation and connect all verification components to a comprehensive verification artifact (Sec. 5).

Our verification is carried out in Coq and utilizes the Verified Software Toolchain (VST [5]) to verify implementation correctness. Our choice of a (dependently-typed) higher-order logic is motivated by the need to integrate multiple abstraction levels in a common logical framework and the need to support the formulation of monotonicity, epistemic principles, and trace-based models. Indeed, prior attempts by Derecho's developers to verify the monotonicity principles in Ivy [27] were unsatisfactory, in part due to a lack of expressiveness. Our choice to use VST is motivated by its foundational connection to a verified compiler [23], its demonstrated robustness and suitability for abstraction-bridging verification, and the fact that it targets an established systems programming language, providing an avenue for future work to gradually integrate treatments of more advanced aspects of Derecho and its applications.

We begin with background on Derecho, Coq, and VST (Sec. 2). Sec. 7 discusses related and future work. A current snapshot of our development, including C code and Coq proof scripts can be found at `https://zenodo.org/records/10819602`.

2 Background on Derecho, Coq, and VST

Derecho. The Derecho system is an open-source library for leveraging high-speed networking in modern cloud computing platforms such as file systems, key-value stores, coordination and pub-sub [17]. The central feature is the support for a strong consistency model: state machine replication (atomic multicast and Paxos, integrated with a "virtual synchrony" layer for self-managed and self-repairing membership). Derecho supports replication and state-machine actions in a manner that maps cleanly to modern communications hardware, as supported by the LibFabric library [12] such as RDMA, DPDK, and other forms of high-speed networking. Even on standard TCP, Derecho is said to be faster than any prior Paxos-like technology [4,39]. In part this speed reflects modern engineering: The system is coded in C++ 17, entirely zero-copy, lock-free, and all data paths are opportunistically batched. The mapping to RDMA is particularly efficient and enables continuous data transmission so that the RDMA transport can run non-stop at its peak speed. Another important aspect of performance is the way that the Derecho protocols are implemented using the *shared state table*, SST. Owing to monotonic operations that can be understood in terms of knowledge, Derecho has a control plane that is especially well-suited to shared memory implemented using one-sided RDMA.

Several industrial applications of Derecho are in deployment or development, for safety- and security-critical applications, which motivates the need for high assurance [39]. It becomes important to formally specify the system itself, to verify that the protocols as implemented are correct solutions with respect to these specifications, to fully quantify their assumptions about the environment and the types of failures that might occur at runtime, and to formally characterize the conditions under which progress can be guaranteed. An initial attempt to prove the Derecho protocols using Ivy [27] was a mixed success [4]. With the help of the Ivy developers, the Derecho team was successful in translating the Derecho specification into a solvable fragment of first order logic and then verifying the protocols using Ivy's prover. But such first-order invariants are not adequate to express the monotonicity and epistemic properties that are key to Derecho and to applications built on it. There has been much other work on verifying Paxos protocols (e.g. [31]) or implementations of Paxos (e.g. [13]) but not on leveraging monotonicity or explicating the role of knowledge. There have been no prior efforts to verify Derecho at the implementation level.

The Coq proof assistant. Coq is a system for interactive development of formal proofs in a powerful higher order logic. The system home page (`coq.inria.fr`) includes information on the user community and teaching materials including the Software Foundations textbook series (`softwarefoundations.cis.upenn.edu`). The Coq system may have been renamed to "The Rocq Prover" by the time the conference takes place.

Verified Software Toolchain. The Verified Software Toolchain (VST) [5] implements a concurrent separation logic for C in Coq and is justified by a machine-checked proof of soundness with respect to the C semantics as formalized by the CompCert compiler [23]. Verification using VST provides higher assurance than SMT-based tools, which rely on trusted axiomatizations of program semantics. Being embedded in Coq, VST specifications can exploit whatever mathematics is convenient for the application domain – including high level distributed system properties.

Similar to auto-active tools like Dafny, the proof engineer using VST must write specs, provide loop invariants, and sometimes guide other reasoning (using

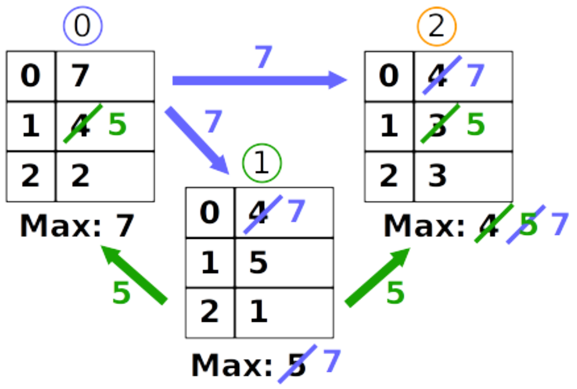

Fig. 1: Distributed system with three nodes, each holding an SST replica and maintaining local state, *Max*. The first column of the SST represents the *myrow* index. Values in the second (data) column monotonically increase over time. When replica 1's row is synchronized with the SSTs of replicas 0 and 2, the local state of replica 2 is updated to value 5. Replica 0's local state is unaffected as the received value does not dominate the current local state. Subsequent synchronization of replica 0's row triggers updates to the local states of replicas 1 and 2, yielding agreement between the local states. A final synchronization of replica 2's row (not shown) would affect the SSTs in replicas 0 and 1 but not the local states.

the Coq IDE). The VST tactics automate forward symbolic execution. Any code verified in VST is proved to be safe and free of undefined behavior, assuring the absence of numeric overflows, memory errors, etc. VST supports refinement ("subsumption") between specifications, so a component can be verified with respect to detailed low-level specs that are in turn connected to more abstract or client-friendly specs, such as a distributed system model.

3 The SST interfaces: code API and trace-based specs

Here is a sketch of the Derecho programming model. A system is comprised of a collection of replicas, as shown in Fig 1. The replicas all run the same program. A replica's state has its private state and its copy of the *shared state table* (SST). Replicas are numbered $0..N-1$ and the SST has N rows. During an *epoch* between failures, there is a fixed group with N members. The shared state of replica i comprises row i, called its *own row*; this is the only row that a replica can write in its SST. The framework asynchronously copies row i from its owner to the SST's of all other replicas. This is called a *sync*. In this way the SST provides replica i with a possibly stale snapshot of the rows for all other replicas j, $j \neq i$.

Fig. 2 shows excerpts from the SST API in C. Function new_system initializes the system, specifying the number nrows of replicas and an application specific

```
typedef struct sst sst_t;
int getMyrow(sst_t *sst);
int getCell(sst_t *sst, int r, int c);
void setCell(sst_t *sst, int c, int z); /* set column c of sst's own row to z */

typedef struct priv_state priv_state_t; /* kept abstract: struct priv_state defined by client */
typedef bool (*predicate_t) (priv_state_t *, sst_t *); /* guard of an action */
typedef void (*trigger_t) (priv_state_t *, sst_t *); /* effect of an action */

typedef struct action action_t;
action_t *mkAction(predicate_t p, trigger_t t, priv_state_t *s);

typedef struct sys_state sys_state_t;
sys_state_t *new_system(int nrows, int nappcols, int *iniRow);
void install_action(sys_state_t *reps, int repId, action_t *act);

void run(sys_state_t *reps, int rounds);
```

Fig. 2: SST interface in C; excerpts.

number nappcols of columns in the SST of each replica. A fixed initial row value iniRow is used to populate all rows of all replicas. The SST is a matrix of **int**'s, but the API can easily be extended to cater for other base types. Finally, in addition to application specific columns, the SST has columns used to store framework specific data. These keep track of whether a replica has been suspected to have failed, or whether a replica is no longer actively participating. Derecho's treatment of failure is important and leverages monotonicity in interesting ways, but for lack of space we do not elaborate on it here.

Application code is organized as a collection of *actions*, where each action comprises a *predicate* and *trigger* in addition to auxiliary private state. (We use the Derecho [17] terms for the guard and effect, respectively, of an action.) The typedefs predicate_t and trigger_t define the type of these as function pointers. A replica's actions can read the SST, and the trigger of an action can write the replica's own row of its SST (function getMyrow returns the index to its own row). The application defines a struct type priv_state_t which is left abstract in the interface. This is meant for the private state used by an action. Each action has its own private state. At a minimum, this can be used to communicate from a predicate (which checks some condition) to its trigger, which gets executed only if the predicate returns true. Private state persists between invocations of an action. Actions are installed during initialization; the client calls install_action for this purpose: the parameter repId is the index of the replica in the system.

In general, application code is structured to include definitions of nrows and nappcols; a definition of priv_state_t; one or more actions, each comprising a predicate and a trigger; and a main function. The main function:

- creates the application-specific data of the initial row,

```
Class ClientParams := { NROWS: Z; NAPPCOLS: Z; privSt: Type; ... }.

Definition repId := {n: Z | 0 ≤ n < NROWS}. (* replica identifier *)
Definition NCOLS := NAPPCOLS + ... (* framework specific columns *)
Definition SSTrow := {xs: list Z | length xs = NCOLS}.
Record SST_Tp := {the_tbl: {xs: list SSTrow | length xs = NROWS}; the_id: repId }.

Definition PredicateTp := SST_Tp → privSt → bool * privSt.
Definition TriggerTp := SST_Tp → privSt → SSTrow * privSt.
Definition ActionTp := PredicateTp * TriggerTp * privSt.

Record replica := {the_sst: SST_Tp; actions: list ActionTp}.
Definition sys_state := {reps: list replica | length reps = NROWS}.
```

Fig. 3: SST model in Coq; excerpts.

- calls new_system to construct and initialize the replicas' SSTs
- allocates private states for the actions,
- calls install_action to install the actions,
- calls run, which runs the main computation,
- and finally deallocates everything, using API functions like teardown.

The main computation is carried out by the function run. Derecho arranges that all the actions of all the replicas are repeatedly invoked. Each invocation first calls the predicate and then, if the predicate returns true, calls the trigger. Function run is called with a bound, rounds, which is the number of rounds to be executed, meaning the number of times a given action is invoked on a given replica.

Coq model and C specifications. Fig. 3 shows types in Coq used to model the SST and replica data structures. A sys_state in Coq is a list of replicas where each replica contains an SST and a list of actions. SSTs are modeled using the type SST_Tp which, essentially, is a matrix of a fixed dimension. The field the_id keeps track of the index of the SSTs own row. Clients specify SST dimensions by creating an instance of the ClientParams typeclass; they also provide the type for private states, provide an initial condition on private states and prove that NROWS and NAPPCOLS fall within implementable bounds (these details are elided in Fig. 3). The types PredicateTp and TriggerTp model predicates and triggers respectively. Note that these types model monadic computations in the state monad for privSt. A trigger returns the updated value for the replica's own row.

The Coq model includes definitions corresponding to each C function in the SST API (Fig. 2). Specifications defined in VSTs separation logic tie C code to model programs. They posit that C implementations have the same effect on concrete structures as model programs have on abstract ones; a connection made precise using representation predicates which relate data structures laid out in the heap in C to structures in Coq. For example, here is the spec for install_action written in stylized notation and using $*$ for separating conjunction.

{ Replicas_rep sys reps ∗ Action_rep act a }
install_action(reps, n, a)
{ Replicas_rep (model_install_action sys n act) reps }

The precondition says the heap contains Replicas_rep sys reps and Action_rep act a. The first conjunct says that pointer reps points to a data structure that represents the abstract system state sys of Coq type sys_state. The second says that pointer a points to a data structure that represents the abstract action act of Coq type ActionTp. Calling install_action then ensures that reps represents an abstract state consistent with applying the model level function model_install_action on the initial abstract state. The postcondition also expresses a transfer of resources: the Action_rep is no longer available to the caller.

Specifications for other API functions are given in a similar style. For predicates and triggers, we define generic specifications. These are used, for instance, by Action_rep act a above which asserts that a's predicate satisfies predicate_spec for the abstract predicate for act (resp. for a's trigger). Here is the generic predicate spec for predicate pred in C with respect to the model predicate model_pred.

{ SST_rep SST sst ∗ privSt_rep pst p }
pred(sst, p)
{ **let** (b, pst') := model_pred SST pst **in** SST_rep SST sst ∗ privSt_rep pst' p ∧ RET b }

The precondition expresses that pointers sst and p represent the abstract SST and the abstract private state pst, respectively. Note that privSt_rep is application-specific and provided to the framework by the client. The postcondition then asserts that the SST is left unmodified and that the effect on the private state is in accord with the function model_pred. Expression RET b denotes the return value, which must also be in accord with model_pred.

The generic trigger spec is similar. Its precondition requires the corresponding predicate to be true and its postcondition ensures that the SST's own row is updated in accord with the given model trigger. Predicates and triggers defined by client applications are required to satisfy these generic specs.

Schedules, traces, and Run spec. In order to capture distributed computation and its inherent nondeterminism, we depart from the above style of specification when it comes to run. The spec for run is formulated in terms of global traces which in turn are based on schedules. A schedule is a list of events that serves to record a linear order in which actions have taken place at particular replicas, and syncs have happened between replicas.

In the model, a sched_item (intuitively, an *event*) is either of the form sch_act r n t which indicates that replica r has performed its nth action at timestamp t, or sch_sync from to t for from≠to which indicates that replica from has sync'd its own row at timestamp t to replica to.[4] Syncs in Derecho are based on totally ordered point-to-point channels (e.g., as implemented in one-sided RDMA) and

[4] In our formal development events are also used to keep track of Derecho-style failures where one replica can suspect another of failing if it hasn't received a sync for some amount of time. We omit discussion since failures are not the focus of this article.

the timestamps associated with events help model this behavior. The timestamp in a sch_sync event connects the sync with the state of its "from" replica following a designated action instance. This is needed because at the time the "to" replica's SST gets updated by the sync, the "from" replica may have changed its state.

We say a list of sched_item's is a schedule if it satisfies certain well-formed constraints on the order of timestamps: (a) action timestamps are unique and increasing, and (b) syncs to a given replica are for actions in increasing timestamp order. This is meant to capture an accurate minimal model of what Derecho provides. Our library defines some additional constraints on schedules, such as bounds on relative progress between actions at different replicas. Such bounds may need to be assumed in order to prove progress properties of applications.

A *trace* is a list of system states generated from an initial system state and a schedule. Traces are defined by the predicate traceOf which goes by induction on the schedule sch, performing state updates in accord with sched_item's in sch. The event sch_act r n t updates replica r's own row in accord to it's nth action. The event sch_sync from to t updates row from in replica to. The updated value is the own row of from at some point earlier in the execution, determined by timestamp t. The model works with finite traces. Although Derecho is a nonterminating system, a trace models execution up to some arbitrary point. The schedule predicate is prefix-closed and likewise the prefix of a trace is itself a trace.

The run API function (Fig. 2) runs a system once its actions have been installed. It acts on a single system state, running actions on all replicas, to simulate a distributed system in which each replica runs concurrently on its own node. In principle, the system runs forever. To reason about partial executions, our prototype parameterizes run on the number of rounds of execution to perform, rounds. The specification for run is given as follows.

```
{ Replicas_rep sys reps ∧ initial sys ∧ Forall (fun r ⇒ length r.(actions) > 0) sys }
run(reps, rounds)
{ ∃sch tr, schedule sch ∧ traceOf sys sch tr ∧ nActions sch sys rounds
  ∧ Replicas_rep (lastState tr) reps }
```

The spec is formulated in terms of the abstract state sys which is required to satisfy an initial condition – this is given by initial sys and includes an application-specific condition provided by clients. The precondition also requires every replica in sys to have at least one action installed. The postcondition says that there is some schedule sch and trace tr of that schedule from initial state sys. Moreover, every action of every replica has been invoked rounds many times as expressed by nActions sch sys rounds. The final conjunct says that the concrete state pointed to by reps represents the final state of the trace.

For invariance-based reasoning about safety properties, this postcondition is directly applicable as we will seen in the example of Sec. 4. Progress-based reasoning is less direct, as it must be in a partial correctness logic like VST. The spec caters for application-specific postconditions of this form: "if sufficiently many action invocations have occurred then [something interesting about the final state]". The antecedent might be expressed by a numerical lower bound on rounds. For an application intended to converge to a result, an alternative to bounding rounds is for the antecedent to say a fixed point has been reached, i.e., the actions no longer change the state. An example of this kind is in Sec. 6.

Monotonicity, system invariants and Knowledge. Invariants on system states are needed in order to prove top-level safety properties of interest. A predicate on system states P is invariant provided it holds initially and is preserved by all transitions of the system. Here, a transition is either a sync from a replica to another, or the invocation of an action on a replica. Showing that P is preserved by syncs may be harder than showing that it is preserved by actions.

An invariance proof is made easier when P is restricted to be of certain special forms. For example, suppose P is of the form "all rows of all replicas satisfy Q", where Q is a predicate on SST rows. To show P is invariant, it suffices to show it holds initially and is preserved by all actions; syncs don't have to be considered at all. This is because such a P is always preserved by syncs. Consider a sync between replica from and to. Preceding the sync, all rows of both replicas satisfy Q. The sync updates row from of replica to. This new row also satisfies Q, and hence Q continues to hold for all rows of both replicas. Our library includes reasoning principles for invariants expressed in these special forms.

Even when P is a predicate on the whole SST or the whole system, one can often reason only in terms of the actions. This can be done when P is monotonic and the actions are *inflationary*, i.e., each action increases the own row of a replica with respect to the pointwise (cell-wise) ordering. In such cases, all traces are non-decreasing. Reasoning based on monotonicity is conceivable for any ordering. However, for our purposes, it suffices to use the magnitude ordering on integers and its pointwise liftings to SST rows, SSTs and system states (all written $\geq$).

Now suppose a sys_state predicate P is monotonic: if P holds for sys, then it holds for any sys' with sys' $\geq$ sys. Then P is stable, provided the actions are inflationary *Stability* of a predicate means that once it holds, it continues to hold. Further, if P is stable and holds in the initial state, then P is invariant. Thus, in a system with inflationary actions, all the monotonic predicates are invariants.

This reasoning is captured by the following two results in our library. File sst_theory.v defines the pointwise ordering on rows, non-decreasingness of traces, etc, and proves this key result.

```
Theorem mono_act_non_dec_trace: ∀(init: sys_state) sch tr,
  allReplicasSame init ∧ inflaActions init ∧ schedule sch ∧ traceOf init sch tr →
  non_dec_trace tr.
```

It says that any trace is non-decreasing provided that the system's actions are inflationary. Traces are based on schedules, so the theorem quantifies over all schedules. It also assumes all replicas have the same own row value initially. This simplification is in accord with our prototype implementation.[5]

The connection with monotonic predicates is made precise by the following theorem: it says that if actions are inflationary and pred is a monotonic predicate

[5] It loses no generality since one of the actions might serve as an actual initializer. Action predicates can be used to arrange that no other actions are enabled until the initializer has run, and that the initializer becomes effectively disabled thereafter.

on SSTs, then if it holds in replica r in the final state of a trace tr', it holds in r at any later final state. (This result can be extended to monotonic predicates on system states.)

Theorem monoPred_stable_trace: $\forall$(pred: SST_Tp $\rightarrow$ Prop) sch ini tr st r,
initial ini $\wedge$ inflaActions ini $\wedge$ monotonic pred $\wedge$ schedule sch $\wedge$ traceOf ini sch tr $\rightarrow$
$\forall$tr', tr' $\lesssim$ tr $\wedge$ pred (lastState tr').[r] $\rightarrow$ pred (lastState tr).[r].

Here $\lesssim$ is the temporal (prefix) order on traces with which we express that trace tr' extends to trace tr. We use notation sys.[r] to refer to replica r in state sys.

These results are used to prove general facts about the SST framework. For example, it's a general invariant in any system with inflationary actions that any row of an SST is less than or equal to the corresponding replica's own row. In this sense, an SST approximates the "ideal SST" comprised of all the own rows. Our library provides general rules for proving invariants of several forms, including those that aren't monotonic.

Prior work on Derecho emphasizes epistemic reasoning [4]. Our library adapts the standard semantics of epistemic logic to our setting. Rather than formalizing the syntax of epistemic logic [10], we define the "knows" operator semantically and prove various useful properties. Here we just sketch the ideas. We consider what a given replica knows, in the final state of a trace, based on what it has observed. The replica observes the sequence of updates to its SST. So two traces are indistinguishable for r if projecting them to the list of r's SSTs, and removing stuttering steps, results in the same list of SSTs. In other words, r can distinguish between two traces only if its SST differs at some step. So r *knows* system predicate P in the final state of tr just if P is true in the final state of all traces r-indistinguishable from tr. This is written Know tr r P. Our library includes a collection of standard theorems about knowledge, as well as connections between knowledge, invariants, and monotonic predicates.

4 Example: stability detection

Our first example, stability_detection, illustrates message streaming. Each replica is equipped with two actions. The first, multicast, sends a message to all replicas in the system. The second, receive, acknowledges receipt of a message. Message sends and receives are modeled using counters, which are kept track of in the SST of each replica.

In the model, sys.[r] is the SST of replica r in state sys, and sys.[r].[i,k] is cell k of row i of replica r. Cell sys.[r].[r,r] stores the number of messages r has sent and sys.[r].[r,k], the number of messages sent by replica k that r has acknowledged. In our setup, r acknowledges a message sent by k simply by incrementing the cell sys.[r].[r,k].

On replica r, the multicast action increments sys.[r].[r,r]. It is guarded by the predicate true, modeling the scenario where message sends are always possible. Other replicas learn about messages from r through syncs of r's own row. The receive action scans r's SST and checks whether there is a pending message from a replica, say k, that hasn't been acknowledged yet. If so, it increments sys.[r].[r,k].

Our model-level implementation of the application first creates an instance of the ClientParams type class, specifying the number of application columns in each SST to be NUM_REPLICAS in accord with the layout described above. Private state in stability_detection is only used by the receive predicate to communicate to its following trigger invocation; it does not rely on the private state persisting between action invocations. (See Sec. 6 for an example using persistence.)

The initial condition asserts that no replica has sent or acknowledged any messages. The implementations of triggers and predicates then use the generic functions from the SST model library to create system states and simulate runs. The model-level application specification is knowledge-oriented:

```
Definition stabDet (tr: list sys_state) : Prop :=
  (∀ (r k:repId), Know tr r (fun sys ⇒ sys.[r].[k,r] ≤ sys.[k].[k,r])) ∧
  (∀ (r k:repId), Know tr r (fun sys ⇒ sys.[r].[k,k] ≤ sys.[k].[k,k])) ∧
  (∀ (i k:repId), everyoneKnows tr (fun sys ⇒
     ForallReplica (fun (r: replica) ⇒ Min_of_col r i ≤ sys.[k].[k,i]) sys)).
```

The first conjunct says any replica r knows it has received no more acknowledgements from k than have been sent by k. The second says any r knows it has acknowledged no more messages from k than have been sent. The third says everyone knows (the $K1$ operator of epistemic logic) for each sender i, receiver k, and every replica r, that k has acknowledged at least Min_of_col r i many messages from i. Thus by computing, on its own SST, the value of Min_of_col r i, replica r can determine which messages from i are committed. All of these properties are relative to a trace tr. The top-level specification, described below, relies on proof that stabDet holds for all traces of an initial system in which the send and receive actions are installed in all replicas.

At the C level, we define predicates and triggers, relying on generic functions provided in simple_sst.h. We prove that each C predicate/trigger pair conforms to corresponding model-level functions, in accord with generic specifications for SST actions defined in our library. For example, we prove the multicast trigger satisfies the generic triggerSpec with its parameters P and T instantiated with predicate **fun** SST st ⇒ (true, st) and the increment of sys.[r].[r,r] respectively.

The top-level program, unittest, calls run for some number of rounds and is structured as a generic application of the SST API (described in Sect. 3). Its specification is as follows.

```
{ Replicas_rep ini reps ∧ initial ini ∧ sys_with_actions ini stabilityActions }
unittest(reps,N)
{ ∃sch tr, Replicas_rep (lastState tr) reps ∧ schedule sch ∧ traceOf ini sch tr ∧ stabDet tr }
```

By formulating this for arbitrary number N of rounds, we show that stabDet holds at any point of any execution.

The code of unittest just calls run, the spec of which provides all but the last conjunct of the postcondition. To prove stabDet tr we reason entirely at the model level: we have a trace in which the multicast and receive actions take place, interleaved with syncs. General results about monotonicity in connection with invariants reduce our proof obligation to reasoning about the two actions.

5 The SST implementation and its verification

Our C implementation represents replicas as an SST table whose main component is row-oriented matrix, and a linked list of actions. Each action contains function pointers for a predicate and a trigger, and some action-private state.

```
struct sst{               struct action{                typedef struct replica{
  int numrows;              predicate_t predicate;        action_t* actions;
  int numcols;              trigger_t trigger;            sst_t sst;} replica_t;
  int my_row;               priv_state_t *act_state;    struct replicas {
  row_t rows[];};           action_t *next;};             replica_t **collection; };
```

We omit descriptions of the administrative functions that construct or deallocate replicas and install or uninstall actions. The main execution loop is the function run, which carries out a client-specifiable number of rounds, each round invoking the following (non-API-exposed) do_actions_sync_all function on all replicas.

```
void do_actions_sync_all(replicas_t* reps, int r)
{ replica_t *rep = reps→collection[r];
  action_t *act = rep→actions;
  while (act)
  { run_action(rep, act);
    for (int to = 0; to < getNumreplicas(reps); to++)
    { if (to != r) sync_sst(&rep→sst, getSST(reps, to)); }
    act = act→next; }}
```

The function traverses the replica's actions by executing run_action on each element – which is non-trivial only in case the action's predicate fires – and immediately invoking sync_sst afterwards to communicate any update to its local SST matrix to all other replicas. We currently implement sync_sst as a memcpy instruction, which suffices for our sequential 'run' function. Future refinements will realize concurrent execution with fine-grained communication primitives (ultimately, RDMA as in Derecho). In anticipation, our specifications – and the model of schedules, traces, etc. – do not expose the sequential nature of our implementation.

The VST specification of do_actions_sync_all asserts adherence to the following Coq expression, where sync_sst_all models the communication and do_action_sys models the execution of a single action:

```
Definition do_actions_sync_all (n:Z) (i:repId) (sys:sys_state) : sys_state :=
let f sys actnum := sync_sst_all NROWS (do_action_sys i actnum sys) i in
fold_left f (upto (Z.to_nat n)) sys.
```

We embed this Coq expression in VST as shown in earlier sections and then verify the C code; we then verify run against the specification from Sec. 3.

6 Example: distributed transitive closure

In our second application, replicas collectively compute the transitive closure of a given graph with N vertices. Each row of the SST is divided into two portions.

```
Definition Find_aux : (Z * Z * Z) → PredicateTp := Fix (well_founded_ltof _ lexSize) ...
Lemma Find_aux_def : ∀(i s e: Z) (SST: SST_Tp) (p: privSt),
  Find_aux (i, s, e) SST p = (* i: intermediate vertex, s: start vertex, e: end vertex *)
    if inBounds (i, s, e)
    then if hasPath SST s i && hasPath SST i e && negb (hasPath SST s e)
         then (true, (i, s, e)) else Find_aux (incr (i, s, e)) SST p
    else (false, (i, s, e)).
Definition Find_pred : PredicateTp := '(i,s,e) ← st_get ;; Find_aux (i,s,e).
Definition Find_trig : TriggerTp :=
  '(i,s,e) ← st_get ;; markPath SST s e ;; st_put (incr (i,s,e)) ;; ret SST.[the_id SST].
```

Fig. 4: The Find action for the distributed transitive closure client.

The first N^2 columns, the edge matrix, stores a copy of the adjacency matrix of the input graph as a one-dimensional array. The second N^2 columns, the path matrix, stores the transitive closure of the graph. The replica is equipped with two actions that update the path matrix. The first, Find, computes new paths in the transitive closure. The second, Copy, learns new paths by copying those that other replicas may have already found. Find suffices to ensure the transitive closure is eventually computed by every replica. Copy improves performance by allowing replicas that are "lagging behind" to catch up.

Implementation of predicates and triggers. The implementation of the Find action at the model level is shown in Fig. 4. It is a rendering of the usual Floyd-Warshall algorithm with two key changes. The first is that it recurses over lexicographically ordered triples of graph vertices; this departs from the usual presentation of the algorithm as three nested loops. This is done by the function Find_aux, elided in Fig. 4. For reasoning, the lemma Find_aux_def provides a convenient unfolding of the definition. This change facilitates the second.

The second change is that the predicate/trigger pair follow a "resumption" style of computation in order to fit with our general framework of SST actions, wherein the trigger is used to update the SST and communication between predicate and trigger is facilitated by private state. The private state for this application stores a triple of graph vertices.[6]

Function Find_pred first checks whether a new path exists via intermediate vertex i between start and end vertices s and e. If so, it updates the private state to (i,s,e). The corresponding path is then marked in the SST by the trigger Find_trig. In addition, the trigger increments the private state so that in a subsequent invocation, the predicate starts searching for paths from the next lexically ordered triple of graph vertices. In the case where a path hasn't been found, the predicate doesn't update the private state and recursively tries the next lexically ordered triple. In Fig. 4, the functions st_get and st_put operate on the state monad. We use standard notations for monads.

[6] This is a simplification for expository purposes. In our implementation, the private state contains an additional component used by the Copy action.

The Copy action is implemented similarly. It continuously polls each row of the SST and marks any path not already recorded in the own row. Both actions monotonically increase the path matrix.

We verify C implementations of the Find and Copy actions using the generic predicate and trigger specs.

Application specifications. Like unittest in Sect. 4, the main program, tc_main, calls run for some number of rounds (here, N^3). Its specification is as follows.

```
{ Replicas_rep ini reps ∧ initial_TC_sys ini gph }
  tc_main(reps)
{ ∃sch tr, let fin := lastState tr in
    Replicas_rep fin reps ∧ schedule sch ∧ traceOf ini sch tr ∧ computedTC fin }
```

In the precondition, initial_TC_sys ini expresses that ini is a system that is initialized with the graph gph and each replica is equipped with the Find and Copy actions. The postcondition says that there is a schedule and a trace and moreover, that each replica has computed the transitive closure of the graph provided it has reached a fixpoint. Since the Find action is sufficient for correctness, we define fixpoint to mean that the Find predicate returns false: there are no more paths to be found in the transitive closure. In the definition below, Find_privSt refers to the private state of the Find action. The antecedent expresses "reached a fixpoint" and the consequent says that the path matrix of sys.[r] is equal to the transitive closure of the edge matrix of sys.[r]. Recall the sys.[r,r] notation refers to the own row (which is at index r) of sys.[r].

```
Definition computedTC sys := ∀r, fst (Find sys.[r] (Find_privSt sys.[r])) = false →
  clos_trans (hasEdge sys.[r,r]) = hasPath sys.[r,r].
```

We verify the final conjunct of tc_main entirely at the model level. We start by proving two invariants of any trace of the application. The first, TC_row_inv, is a predicate on rows and says that the edge matrix of replica sys.[r] is contained in its path matrix, which in turn is contained in the transitive closure of its edge matrix. We prove this using generic lemmas about invariants provided by the SST model library. The second invariant path_via_inv is a predicate on a replica and the private states of its actions. Given sys.[r] and a private state (i,s,e), it says that all paths that go via intermediate vertices {0,...,i} are marked in the path matrix of sys.[r]. This is again proved using library support for reasoning about invariants. Together, these invariants suffice to prove computedTC (lastState tr) holds for any trace tr of the application.

7 Related and future work

TLA+ has been used to verify versions of the Paxos protocol [7]. Padon et al [31] used first-order provers for fully automated verification of several Paxos variants. Ivy [27] has been used to verify single instances of a Paxos interaction [30].

Auto-active verification tools based on SMT typically structure proofs in terms of program annotations (e.g., Dafny, Frama-C, Verifast). IronFleet [13] uses a methodology in which behaviors described in the TLA logic are expressed as Dafny specifications; a version of Paxos, and a sharded key-value store application are verified, and there is an (un-verified) compiler from Dafny to C#.

Like VST, RefinedC [34], Bedrock2 [9], and Autocorres [11] are embedded in higher order proof assistants whose expressive logics enable the integration of code verification and model level reasoning. RefinedC builds on Iris [18] and provides a higher degree of automation thanks to the use of refinement types and backtracking-free proof search. It is based on an adhoc semantics of C rather than an established semantics such as CompCert. Bedrock2 is a C-like language and Coq-verified compiler under active development at MIT. Rather than supporting the entire C language, its language features and compilation strategy are limited, motivated by its intended application domain of embedded systems.

VST's concurrent separation logic enables thread-modular verification of shared-memory concurrent code [24,29] and programs with I/O [46]. The latter work concerns a rudimentary web server in C – one node in a distributed system, whose external behavior is specified using *interaction trees* [44]. The server contains a KV store and communicates using a socket API spec compatible with the implementation-side view of the CertiKOS operating system kernel [26]. Recent work [25] further enhances VST's concurrency capabilities by incorporating the Iris theory and proof mode, atomic specifications [33], higher-order ghost state and invariants, and persistent state. These developments will be relevant for verifying a multi-threaded implementation of our system.

Modeling and reasoning about about state machine replication, failure handling, chain replication, two-phase-commit, and variations of Paxos or Raft protocols in Coq is a topic of substantial current interest (e.g. [41,42,36,16,6]). Most of these works present clean-slate formalizations, proposing novel models or domain-specific proof libraries. Typically, they achieve executability only by extraction to OCaml code (with performance limitations) or handle client-side applications only at the model level.

ADO [15] and Adore [16] are recent Coq theories for justifying state machine replication abstractions, focusing on failures and reconfiguration, respectively. ADO takes inspiration from the push/pull model developed for shared-memory concurrency in CertiKOS and involves an event log trace. It includes an implementation of multiPaxos and Chain Replication, but its execution employs unverified send/recv system calls and applications are directly written inside the Coq prover. Adore enables derivation of ADO's mechanism using a slightly more refined state/event model, but its implementation is only extracted to OCaml and has hence not been subjected to realistic performance evaluation. Likewise for execution in Aneris [19] and Verdi [41] (including its application to Raft [42]) and Disel [35].

Monotonicity-exploiting programming models are increasingly popular, in the context of databases [8], logic programming [1], distributed programming languages [28], and parallel programming [20].

Much recent excitement has centered around monotonic uses of Conflict-Free Replicated Datatypes (CRDTs) [37]. These datatypes feature the ability to merge uncoordinated, concurrent updates across weakly-synchronized replicas, without risking replica divergence. This in turn allows development of software which does not rely on consistent or immediate replication, enabling new trends in everything from local-first software [3] (where a user's machine is not expected to wait for synchronization) to high-speed datacenter settings [43] (where a replica should not need to wait for coordination to make progress).

Recent work has made CRDTs far more tractable as a programming model. Some recent work makes CRDTs easier for programmers to use; in particular, monotonic observations (as proposed in [21]) provide the ability to consistently observe projections of an inconsistently-replicated CRDT. Other work has made CRDTs easier to build; in particular, Katara [22] provides the ability to synthesize CRDTs automatically from programmer-provided specifications.

Future work. We have completed the verification of Derecho's core coordination mechanism together with small applications that embody features of the full replication/consensus protocols. VST ensures only partial correctness but our high level model features partial traces and considers arbitrary finite executions whereby we specify nonterminating computations. We formulate a standard semantics of knowledge, but leave to future work the extension of the model to temporal properties, and its use in verification of the full protocols.

While VST has no intrinsic notion of distributed computation, its separation logic lets us specify a system as a collection of replicas acting in disjoint state spaces, modeling RDMA by `memcpy`. One of our next steps will exploit VST's concurrency principles to verify a more faithful (non-sequential) implementation of Derecho in which each replica executes in a separate thread, with lock-free syncs that exploit monotonicity and the fact that rows are single-writer. Towards verifying the core Derecho group membership and view change protocols, a next step will be to verify a 2-phase commit implementation, building on stability detection. Once the core protocols have been implemented and verified, existing applications built on top of Derecho can be formalized.

Acknowledgments. The authors gratefully acknowledge support from Siemens Corporation for this project, initiated and coordinated by Charif Mahmoudi. First author Ramana Nagasamudram is the beneficiary of a graduate fellowship grant from Siemens called the #FutureMakers Fellowship.

References

1. Arntzenius, M., Krishnaswami, N.: Seminaïve evaluation for a higher-order functional language. Proc. ACM Program. Lang. **4**(POPL), 22:1–22:28 (2020). `https://doi.org/10.1145/3371090`
2. Azmy, N., Merz, S., Weidenbach, C.: A machine-checked correctness proof for Pastry. Sci. Comput. Program. **158** (2018)
3. Bieniusa, A., Haas, J., Kleppmann, M., Mogk, R. (eds.): Programming Local-first Software (ECOOP workshop). European Conference on Object-Oriented Programming (2022), `https://2022.ecoop.org/home/plf-2022`

4. Birman, K., Jha, S., Milano, M., Rosa, L., Song, W., Tremel, E.: Monotonicity and opportunistically-batched actions in Derecho. In: SSS. Lecture Notes in Computer Science, vol. 14310, pp. 172–190 (2023). `https://doi.org/10.1007/978-3-031-44274-2_14`
5. Cao, Q., Beringer, L., Gruetter, S., Dodds, J., Appel, A.W.: VST-Floyd: A separation logic tool to verify correctness of C programs. J. Autom. Reason. **61** (2018)
6. Chajed, T., Tassarotti, J., Theng, M., Kaashoek, M.F., Zeldovich, N.: Verifying the DaisyNFS concurrent and crash-safe file system with sequential reasoning. In: OSDI (2022)
7. Chand, S., Liu, Y.A., Stoller, S.D.: Formal verification of Multi-Paxos for distributed consensus. ArXiv **abs/1606.01387** (2016)
8. Cheung, A., Crooks, N., Hellerstein, J.M., Milano, M.: New directions in cloud programming. In: Conference on Innovative Data Systems Research (CIDR) (2021), `http://cidrdb.org/cidr2021/papers/cidr2021_paper16.pdf`
9. Erbsen, A., Gruetter, S., Choi, J., Wood, C., Chlipala, A.: Integration verification across software and hardware for a simple embedded system. In: PLDI (2021)
10. Fagin, R., Halpern, J.Y., Moses, Y., Vardi, M.Y.: Reasoning About Knowledge. MIT Press (1995)
11. Greenaway, D., Lim, J., Andronick, J., Klein, G.: Don't sweat the small stuff: formal verification of C code without the pain. In: PLDI (2014)
12. Grun, P., Hefty, S., Sur, S., Goodell, D., Russell, R.D., Pritchard, H., Squyres, J.M.: A brief introduction to the OpenFabrics interfaces - a new network API for maximizing high performance application efficiency. In: IEEE Annual Symposium on High-Performance Interconnects. pp. 34–39 (2015). `https://doi.org/10.1109/HOTI.2015.19`
13. Hawblitzel, C., Howell, J., Kapritsos, M., Lorch, J.R., Parno, B., Roberts, M.L., Setty, S., Zill, B.: IronFleet: Proving safety and liveness of practical distributed systems. Commun. ACM **60**(7) (2017)
14. Hellerstein, J.M., Alvaro, P.: Keeping CALM: when distributed consistency is easy. Commun. ACM **63** (2020)
15. Honoré, W., Kim, J., Shin, J., Shao, Z.: Much ADO about failures: a fault-aware model for compositional verification of strongly consistent distributed systems. Proc. ACM Program. Lang. **5**(OOPSLA), 1–31 (2021). `https://doi.org/10.1145/3485474`
16. Honoré, W., Shin, J., Kim, J., Shao, Z.: Adore: atomic distributed objects with certified reconfiguration. In: Jhala, R., Dillig, I. (eds.) PLDI (2022)
17. Jha, S., Behrens, J., Gkountouvas, T., Milano, M., Song, W., Tremel, E., Renesse, R.V., Zink, S., Birman, K.P.: Derecho: Fast state machine replication for cloud services. ACM Trans. Comput. Syst. **36** (2019)
18. Jung, R., Krebbers, R., Jourdan, J., Bizjak, A., Birkedal, L., Dreyer, D.: Iris from the ground up: A modular foundation for higher-order concurrent separation logic. Journal of Functional Programming **28** (2018)
19. Krogh-Jespersen, M., Timany, A., Ohlenbusch, M.E., Gregersen, S.O., Birkedal, L.: Aneris: A mechanised logic for modular reasoning about distributed systems. In: European Symposium on Programming. LNCS, vol. 12075, pp. 336–365 (2020). `https://doi.org/10.1007/978-3-030-44914-8_13`
20. Kuper, L., Turon, A., Krishnaswami, N.R., Newton, R.R.: Freeze after writing: quasi-deterministic parallel programming with lvars. In: POPL. pp. 257–270 (2014). `https://doi.org/10.1145/2535838.2535842`

21. Laddad, S., Power, C., Milano, M., Cheung, A., Crooks, N., Hellerstein, J.M.: Keep CALM and CRDT on. Proc. VLDB Endow. **16**(4), 856–863 (2022). `https://doi.org/10.14778/3574245.3574268`
22. Laddad, S., Power, C., Milano, M., Cheung, A., Hellerstein, J.M.: Katara: Synthesizing CRDTs with verified lifting. Proc. ACM Program. Lang. **6**(OOPSLA2) (2022). `https://doi.org/10.1145/3563336`
23. Leroy, X.: Formal verification of a realistic compiler. CACM **52**(7), 107–115 (2009)
24. Mansky, W., Appel, A.W., Nogin, A.: A verified messaging system. In: OOPSLA (2017)
25. Mansky, W., Du, K.: An Iris instance for verifying Compcert C programs. Proc. ACM Program. Lang. (POPL) (2024)
26. Mansky, W., Honoré, W., Appel, A.W.: Connecting higher-order separation logic to a first-order outside world. In: ESOP (2020)
27. McMillan, K.L., Padon, O.: Ivy: A multi-modal verification tool for distributed algorithms. In: Lahiri, S.K., Wang, C. (eds.) CAV (2020)
28. Milano, M., Recto, R., Magrino, T., Myers, A.: A tour of Gallifrey, a language for geodistributed programming. In: Lerner, B.S., Bodík, R., Krishnamurthi, S. (eds.) 3rd Summit on Advances in Programming Languages (SNAPL 2019). Leibniz International Proceedings in Informatics (LIPIcs), vol. 136, pp. 11:1–11:19. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2019). `https://doi.org/10.4230/LIPIcs.SNAPL.2019.11`
29. Nguyen, D.T., Beringer, L., Mansky, W., Wang, S.: Compositional verification of concurrent C programs with search structure templates. In: CPP (2024)
30. Padon, O.: Deductive verification of distributed protocols in first-order logic. In: FMCAD (2018)
31. Padon, O., Losa, G., Sagiv, M., Shoham, S.: Paxos made EPR: Decidable reasoning about distributed protocols. Proc. ACM Program. Lang. **1**(OOPSLA) (oct 2017)
32. Recio, R.J., Culley, P.R., Garcia, D., Metzler, B., Hilland, J.: A Remote Direct Memory Access Protocol Specification. RFC 5040 (Oct 2007). `https://doi.org/10.17487/RFC5040`, `https://www.rfc-editor.org/info/rfc5040`
33. da Rocha Pinto, P., Dinsdale-Young, T., Gardner, P.: TaDA: A logic for time and data abstraction. In: ECOOP (2014)
34. Sammler, M., Lepigre, R., Krebbers, R., Memarian, K., Dreyer, D., Garg, D.: RefinedC: automating the foundational verification of C code with refined ownership types. In: PLDI (2021)
35. Sergey, I., Wilcox, J.R., Tatlock, Z.: Programming and proving with distributed protocols. Proc. ACM Program. Lang. (POPL) (2017)
36. Sergey, I., Wilcox, J.R., Tatlock, Z.: Programming and proving with distributed protocols. Proc. ACM Program. Lang. (POPL) (2018)
37. Shapiro, M., Preguiça, N., Baquero, C., Zawirski, M.: Conflict-free replicated data types. In: Stabilization, Safety, and Security of Distributed Systems: 13th International Symposium (SSS). pp. 386–400 (2011). `https://doi.org/10.1007/978-3-642-24550-3_29`
38. Sharma, U., Jung, R., Tassarotti, J., Kaashoek, M.F., Zeldovich, N.: Grove: a separation-logic library for verifying distributed systems. In: SOSP (2023)
39. Song, W., Yang, Y., Liu, T., Merlina, A., Garrett, T., Vitenberg, R., Rosa, L., Awatramani, A., Wang, Z., Birman, K.P.: Cascade: An edge computing platform for real-time machine intelligence. In: ApPLIED '22 (2022)
40. Timany, A., Gregersen, S.O., Stefanesco, L., Hinrichsen, J.K., Gondelman, L., Nieto, A., Birkedal, L.: Trillium: Higher-order concurrent and distributed separation logic for intensional refinement. Proc. ACM Program. Lang. (POPL) (2024)

41. Wilcox, J.R., Woos, D., Panchekha, P., Tatlock, Z., Wang, X., Ernst, M.D., Anderson, T.E.: Verdi: a framework for implementing and formally verifying distributed systems. In: PLDI (2015)
42. Woos, D., Wilcox, J.R., Anton, S., Tatlock, Z., Ernst, M.D., Anderson, T.E.: Planning for change in a formal verification of the raft consensus protocol. In: CPP (2016)
43. Wu, C., Faleiro, J.M., Lin, Y., Hellerstein, J.M.: Anna: A KVS for any scale. IEEE Transactions on Knowledge and Data Engineering **33**(2), 344–358 (2021). https://doi.org/10.1109/TKDE.2019.2898401
44. Xia, L., Zakowski, Y., He, P., Hur, C., Malecha, G., Pierce, B.C., Zdancewic, S.: Interaction trees: representing recursive and impure programs in Coq. Proc. ACM Program. Lang. **4**(POPL) (2020)
45. Zave, P.: Reasoning about identifier spaces: How to make Chord correct. IEEE Trans. Software Eng. **43**(12), 1144–1156 (2017)
46. Zhang, H., Honoré, W., Koh, N., Li, Y., Li, Y., Xia, L.Y., Beringer, L., Mansky, W., Pierce, B., Zdancewic, S.: Verifying an HTTP Key-Value Server with Interaction Trees and VST. In: ITP (2021)