ShadowDB: A Replicated Database on a Synthesized Consensus Core

Nicolas Schiper, Vincent Rahli, Robbert Van Renesse, Mark Bickford, and Robert L. Constable

Cornell University

Abstract

This paper describes ShadowDB, a replicated version of the BerkeleyDB database. ShadowDB is a primary-backup based replication protocol where failure handling, the critical part of the protocol, is taken care of by a synthesized consensus service that is correct by construction. The service has been proven correct semi-automatically by the Nuprl proof assistant. We describe the design and process to prove the consensus protocol correct and present the database replication protocol. The performance of ShadowDB is good in the normal case and recovering from a failure only takes seconds. Our approach offers simplified means to diversify the code in a way that preserves correctness.

1 Introduction

Building high-assurance distributed systems is a notoriously difficult task. A small bug in the code can sometimes impact millions of users. In April 2011, Amazon's storage service suffered a major outage caused by a router misconfiguration and an improperly handled race condition. It took more than 12 hours for the engineers to contain the problem. Examples in replication software also exist. One of Google's implementation of Paxos [8], a protocol to make services highly-available using state machine replication, broke one of the protocol's invariants leading to potential diverging replicas, in the hope of allowing servers with corrupted disks to recover.

A natural question to ask is how can we build reliable distributed systems that are less prone to bugs and incorrect optimizations. One possible answer is to resort to model checking to ensure the correctness of a protocol. This strategy has been used to model check Paxos as well as other consensus algorithms [16, 22]. However, this approach has two main limitations. The state space of the protocol is often trimmed by checking only a subset of the possible process interleavings and only a subset of the variable states is considered. Second, it is rarely the actual code that is model checked but rather a high-level specification of the protocol. One must be careful not to introduce bugs in the protocol implementation, a difficult task when the software is complex and distributed.

To remedy these shortcomings, we propose to identify the critical components of a distributed system and synthesize *correct-by-construction* code for these components. In distributed protocols, the code handling normal case operations is usually easy to debug and well-tested. The code handling failures is more complex and much harder to test however. This paper is a progress report of the implementation of a replicated version of BerkeleyDB where failure handling is taken care of by a synthesized consensus protocol. In ShadowDB, replication is done in a primary-backup fashion: one database is designated the primary and defines the order in which transaction updates are applied to the backup databases. When failures occur, consensus is used to agree on a new replica configuration.

The consensus protocol was specified, proven correct, and executable code synthesized in the Nuprl proof assistant [11, 2]. Nuprl allows distributed protocols to be specified in the EventML functional language [6, 21]. Proving theorems about the protocol, such as agreement on the decided value in the case of consensus, is semi-automatic: some theorems are proven manually while others are generated and proved automatically. Given an EventML specification, Nuprl can also synthesize code that is guaranteed to satisfy the specification. The code is synthesized as a Nuprl term, and runs inside an evaluator. Once the correctness proofs are complete, the synthesized code is correct-by-construction and thus bug-free w.r.t. its specification and the correctness criteria.

Nuprl implements constructive type theory [2] in which programs can be *extracted* from proofs, thereby creating programs that satisfy the specifications given by theorems. Given a specification, the code synthesized by Nuprl is extracted from a proof that the specification is implementable by a collection of distributed processes.

Several other systems that are based on constructive logic also feature program extraction. A prominent example of such a system is Coq [3, 1, 18]. Using Coq, Leroy [17] has developed and certified a compiler for a C-like language. The compiler was obtained using Coq's extraction mechanism to Caml code.

Formal program synthesis from formal specifications is a robust and flexible approach to build correct and easy to maintain software. For example, Kestrel Institute synthesizes concurrent garbage collectors by refinements from specifications [20, 19]; Bickford et al. [10, 7] synthesize correct-by-construction distributed programs from specifications written in a theory of events.

Our approach offers an important feature: it allows synthesizing diversified versions of the replicas. For instance, the synthesized code can use different container data structures (e.g., hash tables or binary trees), and the code evaluator can be written in different languages. We currently have evaluators in OCaml and SML. In doing so, the various replicas interact with their respective software and hardware environments in different ways and are more likely to fail independently.

The rest of this paper is organized as follows. We present the system model and some definitions in Section 2. Nuprl and the synthesized consensus protocol are presented in Section 3. We describe ShadowDB and how it integrates the consensus core in Section 4. Performance results of ShadowDB are reported in Section 5. We conclude the paper in Section 6.

2 Model and Definitions

We assume an asynchronous message-passing system consisting of processes, some of which may crash. When a process crashes, it stops its execution permanently. A process that never crashes is said to be correct.

There is no bound on the time it takes for a process to execute the next step of its code nor on messages to be received. The system is fair however: given enough time, any process running on a correct machine takes at least one step and if a message is repeatedly sent by a correct process to a correct process, then that message is eventually received.

Consensus allows processes to propose a value and ensures that a single proposal is eventually selected [12]. Consensus is defined by three properties: (i) if a process decides on v, then v was proposed by one of the processes (validity), (ii) all processes that decide, decide the same value (agreement), (iii) all correct processes eventually decide (termination). By iteratively running instances of consensus, one can build a total order of decisions.

No consensus protocol can be live in a purely asynchronous system with crash failures [13], we thus make extra assumptions about the system's synchrony. We suppose that the network goes through *good* and *bad* periods. In bad periods, messages may experience large delays, be lost, and processes may be slow. In good periods, correct processes communicate in a timely fashion. There are infinitely many *long enough* good periods for consensus to decide on a value and for ShadowDB to execute a transaction.

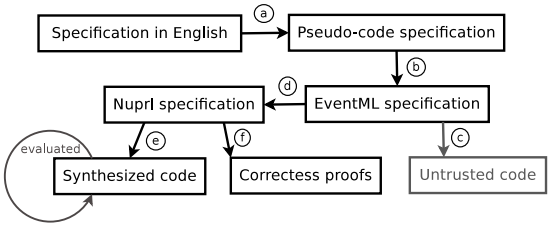

Figure 1: Synthesis and verification workflow in Nuprl.

ShadowDB offers a transactional key-value store interface and ensures that the execution is linearizable [15]: transactions appear to have been executed sequentially on one database at a single point in time between the start of the transaction at the client and the reception of the transaction's response. Clients issue multiple put and get operations per transaction by which they respectively retrieve and update the value of a given key. Both keys and values are arbitrarily long byte strings.

3 Formal Synthesis of Consensus

3.1 Methodology. Figure. 1 illustrates our workflow to obtain correct-by-construction code from high-level specifications. Given a protocol specification (e.g., pseudo-code), we generate by hand a corresponding EventML specification (see markers ⓐ and ⓑ in Figure 1). EventML is an ML [14] dialect targeted to develop networks of distributed processes that react to *events*. An event is an abstract object corresponding to a point in space/time that has information associated to it. Concretely, an event can be seen as the receipt of a message by a process. Using EventML, programmers define distributed systems by specifying their information flow, and the tool automatically synthesizes code from such specifications. EventML features a small set of logical combinators allowing one to build systems by combining together smaller pieces, i.e., small processes in charge of achieving simple tasks.

Even though EventML can synthesize highly reliable code without having to interact with Nuprl, this code is not formally guaranteed to be correct (see marker ⓒ in Figure 1). This untrusted code can be used for testing and debugging purposes before starting the formal verification of the specified protocol's correctness. In order to obtain correct-by-construction code, one has to interface with Nuprl. EventML facilitates this interaction by generating the formal semantic meaning of specifications (see marker ⓓ in Figure 1), i.e., predicates expressed in

```
process round (replicas, f, slot, val, i)
  var voters = ∅, vals = [];
  ∀ r ∈ replicas :
       send(r,⟨``vote``,(((slot,i),val),self())⟩);
  for ever
    switch receive()
      case ⟨``vote``,(((slot,i),c),sender)⟩ :
        if sender ∉ voters then
          voters := voters ∪ {sender};
          vals   := c . vals;
        end if;
        if length vals = 2 * f + 1 then
          let (k,x) = possmaj vals c in
          if k = 2 * f + 1 then
            send(self(),⟨``decided``,(slot,x)⟩);
          else
            send(self(),⟨``retry``,((slot,i+1),x)⟩)
          end if;
          exit();
        end if;
      end case
    end switch
  end for
end process
```

```
(* -- state update -- *)
let add_to_quorum si loc ((si',v),sender) (vals,locs) =
  if si = si' & !(deq-member (op =) sender locs)
  then (v.vals, sender.locs)
  else (vals,locs);;
(* -- state -- *)
class QuorumState si =
  State1 (\loc.([],[])) (add_to_quorum si) vote'base;;
(* -- output -- *)
let when_quorum si loc (((s,i),v),sender) (vals,locs) =
  if si = (s,i) & !(deq-member (op =) sender locs)
  then if length vals = 2 * f + 1
       then let (k,x) = possmaj valeq vals v in
            if k = 2 * f + 1
            then { decided'send loc (s, x) }
            else { retry'send loc ((s,i+1), x) }
       else {}
  else {};;
(* -- Quorum and Round -- *)
class Quorum si =
  (when_quorum si) o (vote'base, QuorumState si);;
class Round (si,c) =
    Output(\loc.vote'broadcast reps ((si,c),loc))
  || Once(Quorum si);;
```

Figure 2: A round of 2/3 consensus in pseudo-code (left) and EventML (right).

the Logic of Events [4, 5] which is a logical framework implemented in Nuprl to reason about and synthesize distributed systems. EventML can be seen as a programming interface to the Logic of Events.

Once the formal semantic meaning of a specification is loaded in Nuprl, we have designed a tactic that automatically proves that the specification is implementable, i.e., that there exists a collection of distributed processes that produce exactly the information flow defined by the specification. That tactic works by recursively proving that each component of the specification is implementable. Because EventML specifications only use a small number of combinators, which we have proved to be implementable, proving that a specification is implementable is fairly straightforward. One can then extract from such a proof the above mentioned collection of processes that is provably correct w.r.t. the specification (see marker ⓔ in Figure 1). The synthesized code is a collection of Nuprl terms that can be evaluated using any of the term evaluators available in EventML. Because processes react to and produce messages, EventML also features a message system. Finally, the last step of our methodology (see marker ⓕ in Figure 1) consists in proving that the specified protocol satisfies some correctness criteria, such as the consensus properties. Our logical framework allows to do that reasoning at the specification level rather than at the code level. Event though Logic of Events and EventML specifications might look like programs more than high-level specifications, the level of abstraction of the Logic of Events allow us to easily perform both synthesis and verification. For that reason we refer to Logic of Events and EventML pieces of code as either specifications or programs.

3.2 Synthesis and Verification of Consensus. We illustrate our approach using the *round* process, a key component of the consensus protocol we have synthesized, denoted 2/3 consensus [9]. This protocol tolerates up to f crash failures, by using at least $3f+1$ replicas. Using this protocol, decisions can be reached in one round, compared to protocols that use $2f+1$ replicas and need at least two rounds to decide.

Figure 2 shows pseudo-code for a round, a sub-process spawned by a replica, running at the same location as the replica. A round tries to decide on a value for a given consensus instance, also called a slot. Once it is done, it sends a report back to its parent process (either a "decided" or a "retry" message) and exits.

A round process takes five parameters: replicas is the collection of replicas trying to reach consensus; f is the number of tolerated failures; slot is the slot number that round is trying to fill with value val; finally, because a round might not succeed, replicas allow arbitrarily many do-over polls: successive polls for a given slot number are assigned consecutive integers called innings. A round process is in charge of a single inning.

A round R proceeds as follows: First, R sends votes for (((slot , i), val), self ()) to each of the replicas, where self () is R's identifier. Then, R waits for (2∗f)+1 votes for slot number slot and inning i from the collection of replicas. If the (2∗f)+1 votes are unanimous (this is computed by possmaj), R send a "decided" message to its parent process (i.e., to itself), otherwise it sends a "retry" message to its parent process.

From the pseudo-code displayed in Figure 2, we have generated by hand the corresponding EventML specification presented on the right of the same figure (our EventML tutorial [6] presents the complete specification). A Round broadcasts a vote to the replicas and runs what we call a Quorum process. Once that process has produced an output it exits, as specified by the keyword Once. A Quorum reacts to "vote" events and maintains a state where it stores the votes it has received.

Once our specification was loaded in Nuprl, we have synthesized code that provably implements it, and we proved that the safety properties of 2/3 consensus hold, namely agreement and validity. Agreement implies that notifications (notifications are sent by replicas upon receipt of "decided" messages) never contradict one another. We proved in Nuprl the following Logic of Events statement saying that if at two events e_1 and e_2, two processes receive two notifications that values v_1 and v_2 both have to fill slot number n, then $v_1 = v_2$:

$$\forall e_1, e_2 : \mathtt{E}.\ \forall n : \mathbb{Z}.\ \forall v_1, v_2 : \mathit{Value}.\ \left(\begin{array}{ll} & \langle n, v_1 \rangle \in \mathit{notify'base}(e_1) \\ \wedge & \langle n, v_2 \rangle \in \mathit{notify'base}(e_2) \end{array} \right) \Rightarrow v_1 = v_2$$

For validity, we proved that the following Logic of Events statement is true, i.e., if at event e a process receives a notification that value v has to fill slot number n, then the pair $\langle n, v \rangle$ has been proposed causally before e:

$$\forall e : \mathtt{E}.\ \forall n : \mathbb{Z}.\ \forall v : \mathit{Value}.\ \langle n, v \rangle \in \mathit{notify'base}(e) \Rightarrow \downarrow \exists e' : \mathtt{E}.\ e' < e \wedge \langle n, v \rangle \in \mathit{propose'base}(e')$$

Proving these two safety properties involved automatically generating and proving thirteen lemmas, and proving by hand eight other lemmas (three of them being trivial, and therefore candidates for future automation).

Termination of 2/3 consensus holds when, in a round, correct replicas receive votes for the same value from a quorum, an assumption likely to hold in a LAN where messages are often spontaneously ordered (proving that property is left for future work).

4 ShadowDB

ShadowDB is a primary-backup replication protocol: a designated primary executes client transactions, sends the transaction updates to the backups, and answers the client with the transaction's outcome once all backups have acknowledged reception of the transaction. ShadowDB relies on the synthesized 2/3 consensus protocol to reconfigure the set of replicas when failures occur. We first explain ShadowDB when there are no failures and then explain how failures are handled.

4.1 The Normal Case. Figure 3 illustrates the ShadowDB protocol. To tolerate f failures among the database servers, we use one primary and f backups. Processing a transaction T sent by a client happens in eight steps as follows:

1. The client sends T to the database it believes to be the primary. If the contacted database is not primary, it redirects the client to the current primary. If the client does not receive any answer after some time, the client contacts another database.

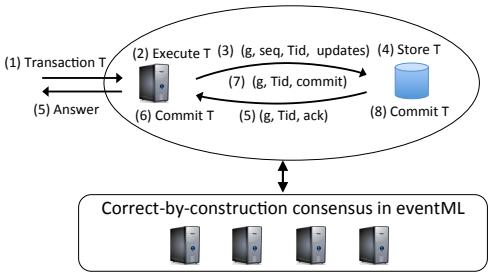

Figure 3: The ShadowDB protocol with $f = 1$.

2. The primary executes T upon first reception. If T has been executed already, the result is communicated back to the client (not shown in Figure 3). Otherwise, T is executed, its updates are extracted, and T is assigned a sequence number—the index of T in the total order of transaction updates. Committing T at the primary is delayed until all backups have replied.
3. The *group incarnation number*, identifying the replica group's membership (explained below), T's sequence number, unique identifier, and updates are sent to all backups.
4. Once a backup receives T, it checks whether its group incarnation number is identical to the one received and T's sequence number corresponds to the next sequence number the backup must handle—this ensures that updates are applied in the same order at backups. If this is the case, the received message is stored locally, and the backup acknowledges reception of the transaction to the primary.
5. The primary waits to receives an acknowledgement from all f backups. When this is the case, the primary knowns the transaction cannot be lost and the primary thus commits T locally.
6. An answer is transmitted to the client. This answer contains the transaction's result, if any, and notifies the client of the transaction's commit.
7. The primary notifies backups that T can commit.
8. Upon receiving such a message, backups apply T's updates and commit T. If T is read-only, this is a *no-op*—read-only transaction must nevertheless go through steps 1-7 of the protocol to ensure they read up-to-date state.

In our current implementation of ShadowDB, the primary and backups execute the above protocol for batches of transactions. This greatly improves throughput as transactions can be committed and persisted to disk in groups. A second optimization is to piggyback the commit message sent to the backups on the next transaction updates the primary sends. This is another form of batching at the message level.

4.2 Handling Failures. In the case of failures, the presented protocol is not live. A failing primary prevents clients from submitting transactions; failing backups pre-

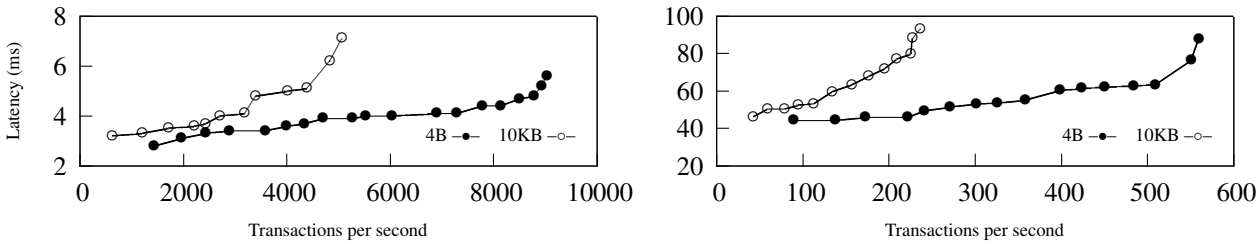

Figure 4: The performance of ShadowDB with read-only (left graph) and update transactions (right graph).

vent the primary from collecting sufficiently many acknowledgments. To remedy this situation, the group of replicas is reconfigured.

When the primary suspects one of the backups to have crashed, or similarly, when a backup suspects the primary to have crashed, the corresponding process proposes its identifier, a new group configuration for which it is the primary, and the last sequence number to consider for the current configuration, to the next instance of consensus that has not decided on a value yet. The proposing process stops processing transactions in the current configuration, ensuring that no more transactions can commit before the reconfiguration is complete. Transactions that were executing at the primary are locally aborted—these transactions will either be retransmitted by the clients, if the current primary is replaced by a new one, or they will be re-executed by the same primary. Consensus ensures that databases agree on the sequence of group reconfigurations, and avoids situations where two distinct replica groups would be formed due to network partitions, leading to diverging database states.

Once consensus decides on the next group configuration, the new primary commits all transactions that it has acknowledged but not committed yet (the new primary may have been a backup in the previous configuration). This ensures that we consider all transactions whose answer has been communicated to the client, and perhaps more. The primary then sends a snapshot of its database to all backups along with the sequence number consensus decided on incremented by one, and the next group incarnation number. This snapshot is handled as a transaction would be in the normal case.

If crashes or failure suspicions happen during the reconfiguration, the replica group is reconfigured again. The procedure will succeed once the system is in a good period for long enough and no crashes occur.

5 Evaluation

The implementation of ShadowDB consists of two parts: the consensus synthesized code, interpreted by an SML program, and the database replication protocol, written in Java. The two components communicate using sockets. Each database node runs a local copy of BerkeleyDB. In the current prototype of ShadowDB, the primary does not extract updates and transactions are simply re-executed at the backups. The primary database monitors backups and backups monitor the primary by periodically exchanging heartbeat messages at a configurable rate. If some database a has not received any heartbeat or application messages from some database b within some user-specified amount of time, database a suspects b to have crashed.

We evaluated ShadowDB on a cluster of 8 Linux machines connected by a gigabit switch. The consensus service ran on 4 quad-core 3.6 Ghz Intel Xeons, the databases and clients ran on dual-core 2.8 Ghz AMD Opterons. The machines were equipped with 4GB of RAM. We set $f = 1$, and ShadowDB consisted of two databases. We used a third database in experiments where we crashed the primary. Clients were running on a separate machine.

We considered two workloads consisting of read-only transactions and two consisting of update transactions. Read-only transactions read either 4 bytes (B) or 10 kilobytes (KB) of data, and update transactions read and wrote 4B or 10KB of data. In all cases, database keys were integers. BerkeleyDB was configured to commit transactions to disk synchronously.

5.1 Normal Case. Figure 4 presents the average latency to commit a transaction as a function of the average number of committed transactions per second. Each experiment comprised between 2 and 42 clients.

The left graph considers read-only transactions that respectively read 4B and 10KB, the right graph considers the same value sizes for update transactions. Not surprisingly, ShadowDB scales better when accessing less data. The latency of update transactions is mainly governed by the time taken to commit a transaction to disk: before answering the client, the primary must send the transaction's update to the backups, the backups commit the previous transaction to disk (recall that commit messages are piggybacked on the next transaction sent to the backups), and the primary commits the transaction after receiving acknowledgments from all backups. Since committing a transaction to disk took about 20 milliseconds, this leads to a minimum latency of 40 milliseconds.

5.2 Recovery. In Figure 5, we illustrate an execution of ShadowDB with a crash of the primary, and measure the execution speed of the consensus service. We consider update transactions that access 10KB values and 5 clients. The execution started with two databases in the replica group, the third database remained idle until the group was reconfigured. After 21 seconds of execution, we simulated a crash of the primary by stopping the corresponding process. We set the heartbeat timeout to 5 seconds, and detected the crash of the primary 4.5 seconds after its actual crash. The backup database proposed a new group configuration that included itself as the new primary and the third database as the new backup. It took 3.5 seconds for the consensus service to decide on the proposed configuration. After the new primary transmited its database snapshot to the new backup, clients connected to the new primary and resumed their execution at time 36 seconds.

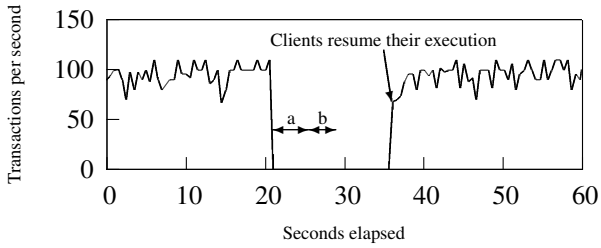

Figure 5: An execution with a crash of the primary (a: crash detection time of 4.5 sec, b: reconfiguration time of 3.5 sec).

6 Conclusion

This paper presented ShadowDB, a replicated database that relies on a synthesized consensus service to handle failures. ShadowDB offers good performance in the normal case and took about 3.5 seconds to reconfigure the replica group. We are currently working on applying standard compilation optimization techniques to render the synthesized code faster.

One could argue that our approach does not result in more reliable code as there might be bugs in Nuprl. Although we recognize that this could be an issue, Nuprl is based on the LCF tactic mechanisms [14], and is especially safe because Nuprl's primitive proofs were checked by an independent prover, ACL2. The extractor and evaluator are simple code, and we could write separate verifications for those, but in 20 years of using the code, we have not found a problem.

Our future plans are to introduce diversity in the consensus service by periodically changing the consensus protocol, one of which will be Paxos, and to synthesize parts of the normal case protocol of ShadowDB.

References

[1] The Coq Proof Assistant. `http://coq.inria.fr/`.

[2] S. F. Allen, M. Bickford, R. L. Constable, R. Eaton, C. Kreitz, L. Lorigo, and E. Moran. Innovations in computational type theory using Nuprl. *J. Applied Logic*, 4(4):428–469, 2006.

[3] Y. Bertot and P. Casteran. *Interactive Theorem Proving and Program Development*. SpringerVerlag, 2004.

[4] M. Bickford. Component specification using event classes. In *Component-Based Software Engineering, 12th Int'l Symp.*, volume 5582 of *Lecture Notes on Computer Science*, pages 140–155. Springer, 2009.

[5] M. Bickford and R. L. Constable. Formal foundations of computer security. In *NATO Science for Peace and Security Series, D: Information and Communication Security*, volume 14, pages 29–52. 2008.

[6] M. Bickford, R. L. Constable, R. Eaton, D. Guaspari, and V. Rahli. *Introduction to EventML*, 2012. `www.nuprl.org/software/eventml/IntroductionToEventML.pdf`.

[7] M. Bickford, R. L. Constable, J. Y. Halpern, and S. Petride. Knowledge-based synthesis of distributed systems using event structures. *Logical Methods in Computer Science*, 7(2), 2011.

[8] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: an engineering perspective. In *Proc. of the 26th ACM Symp. on Principles of Distributed Computing*, pages 398–407, Portland, OR, May 2007. ACM.

[9] B. Charron-Bost and A. Schiper. The heard-of model: computing in distributed systems with benign faults. *Distributed Computing*, 22(1):49–71, 2009.

[10] R. Constable, M. Bickford, and R. van Renesse. Investigating correct-by-construction attack-tolerant systems. Technical report, Department of Computer Science, Cornell University, 2010.

[11] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing mathematics with the Nuprl proof development system*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986.

[12] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, April 1988.

[13] M. J. Fischer, N. A. Lynch, and M. S. Patterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, April 1985.

[14] M. J. C. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation.*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[15] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *Trans. on Programming Languages and Systems*, 12(3):463–492, July 1990.

[16] L. Lamport. Fast Paxos. *Distributed Computing*, 19(2):79–103, 2006.

[17] X. Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *33rd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 42–54. ACM, 2006.

[18] P. Letouzey. Extraction in Coq: An overview. In *Logic and Theory of Algorithms, 4th Conf. on Computability in Europe, CiE 2008*, volume 5028 of *Lecture Notes on Computer Science*, pages 359–369. Springer, 2008.

[19] D. Pavlovic, P. Pepper, and D. R. Smith. Formal derivation of concurrent garbage collectors. In *Mathematics of Program Construction, 10th Int'l Conf., MPC 2010*, volume 6120 of *Lecture Notes on Computer Science*, pages 353–376. Springer, 2010.

[20] D. Pavlovic and D. R. Smith. Software development by refinement. In *Formal Methods at the Crossroads. From Panacea to Foundational Support, 10th Anniversary Colloquium of UNU/IIST*, volume 2757 of *Lecture Notes on Computer Science*, pages 267–286. Springer, 2002.

[21] V. Rahli. Interfacing with proof assistants for domain specific programming using EventML. Accepted to UITP 2012.

[22] T. Tsuchiya and A. Schiper. Using bounded model checking to verify consensus algorithms. In *DISC*, pages 466–480, 2008.