A Diversified and Correct-by-Construction Broadcast Service

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

Cornell University, Computer Science Department

Abstract—**We present a fault-tolerant ordered broadcast service that is correct-by-construction. Our broadcast service allows for diversity *in space*, whereby the participants in the broadcast protocol run different code, as well as *in time*, whereby the protocol itself is changed periodically. We use the Nuprl proof assistant to specify the service, prove correctness, and synthesize the code. The paper includes initial performance results.**

I. Introduction

An application can be made fault-tolerant by replicating it. It is then important that the replicas receive the same inputs in the same order. This is known as the Replicated State Machine Approach [25], [35]. In this paper we present Aneris[1], a fault-tolerant ordered broadcast service that provides replicas with the same input streams. Aneris itself needs to be replicated in order to survive failures. For maximum robustness, it is important that the replicas fail independently.

In a typical implementation of a replicated service, all participants run the same code and share vulnerabilities in that code. Also, the broadcast service runs continuously, allowing adversaries unbounded time to compromise the service. In order to protect the service, it is not only important that participants run different code and maintain different representations of state, but also that this code and state changes over time proactively. This is known as *moving target defense* [23].

Achieving such diversity can only be done automatically, that is, through synthesis. This paper presents early experiments with doing just this. We synthesize intermediate correct-by-construction code from specifications. That code is then evaluated by the participants in an environment that is by-and-large hand-written and may have various bugs. This environment includes the hardware, the operating system that runs on that hardware, the compiler that compiles the code, and the run-time and libraries of the language environment. By diversifying the code we synthesize, we make it likely that vulnerabilities in the environment are triggered independently in different participants. By updating the code over time, we make it harder for an adversary to find and exploit those vulnerabilities.

We use the Nuprl proof assistant [16], [2] to specify Aneris, prove correctness, and synthesize code. Nuprl allows distributed protocols to be specified in the EventML functional language [33]. 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], [16] 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, rather than from a proof that the specification is *correct*. Proofs of correctness are done independently as explained in Sec. V. Thus the proof is a high level version of the program, and there can be several variants of it at this level.

Aneris uses a *consensus protocol* to agree on the order of input messages to the application replicas and is able to switch consensus protocols. We have synthesized code for two crash-resilient protocols: Paxos [26] and a protocol we call 2/3-consensus [13]. Paxos is widely used and has many deployed implementations: Google's Chubby lock service [11] and database Megastore [3], Microsoft's Autopilot service [22], and the Ceph distributed file system [39]. Diversity in time is accomplished by switching between consensus protocols. Diversity in space is provided by running synthesized code in different evaluators, one implemented in OCaml and one in SML. Although not implemented yet, we plan to provide more diversification by synthesizing code that uses different data structures and sends diversified messages between replicas of the broadcast service.

The rest of this paper is organized as follows. Sec. II reviews the related work. Sec. III discusses the system model. Sec. IV presents the specification of the ordered broadcast service. Sec. V describes the methodology to synthesize and guarantee correctness of the service. Sec. VI presents preliminary performance results of our service, and Sec. VII concludes.

II. Related Work

N-version programming [14] is the first technique that argues that for fault-tolerance not only redundancy but also

[1]In Discordian mythology, Aneris is the Goddess of order.

diversity is needed. However, it has been shown that different programming teams introduce correlated bugs into the code of the same specification [24]. Also, while the methodology may provide diversity in space, it does not scale for diversity in time. While our synthesized approach has promise to do better on both these fronts, we have yet to perform tests to show that a synthesized approach provides superior failure independence.

Various projects have support for dynamic protocol updates [20], [21] or study how protocols should be adapted [12], [10]. In [38], [34], the authors consider a stack of network protocols where each layer of the stack can be dynamically changed. The papers list properties that are maintained by the protocol stack when one of the layers is changed, assuming that the protocols themselves are correct.

Bickford et al. [9], [8] developed a switching protocol that allows changing a communication protocol on the fly while maintaing some of its properties (e.g., ordered delivery). They proved its correctness, and they characterized those communication properties that can be preserved by switching as well as those invariants that the switching protocol must satisfy to work correctly. They envisioned that such systems "will be able to increase security at run-time, for example when an intrusion detection system notices unusual behavior." [9]. However, they did not have the formal infrastructure to do synthesis at that time.

There exist other systems based on constructive logic that feature program extraction. Nuprl [16], [2] was the first and Coq [4], [1] is another closely related and very prominent system supported by INRIA.. Using Coq, Leroy has developed and certified a compiler for a C-like language [30]. The compiler was obtained using Coq's extraction mechanism to Caml code. The compiler is certified thanks to "a machine-checked proof of semantic preservation" [30]. Our approach is similar to Leroy's in the sense that extraction does not immediately imply correctness of the extracted program. Correctness is obtained via accompanying formally proved statements that the program satisfies desirable properties.

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 [32], [31]. High-level specifications are stated in terms of "high-level abstract data structures" [31] which get refined to concrete data structures. In contrast, our specifications are based on concrete data structures (raising the level of abstraction of our specifications is left for future work). Also, Bickford et al. [15], [7] synthesize correct-by-construction distributed programs from specifications written in a theory of events.

Some efforts have relied on model-checking to verify the correctness of Paxos [27] as well as other consensus protocols [37]. The main drawback of these approaches is that consensus specifications are checked rather than actual code, and there is no formal mapping from specification to running code.

Recently, Lamport proved the safety of a Byzantine Paxos algorithm in TLAPS [28], [29] using a chain of refinement mappings from Byzantine Paxos to a "trivial, high-level specification of consensus" [29]. Using these refinement mappings, proving that Byzantine Paxos is safe boils down to proving the safety of the trivial consensus specification. However, TLAPS does not perform program synthesis.

F* [36] is a dependently-typed programming language which aims, among other things, at verifying protocols using types (such as refinement types) expressive enough to formalize and reason about security properties.

III. Model and Definitions

We assume a distributed system where processes may fail by crashing. A process that crashes stops its execution permanently and stops sending messages. A process that never crashes is said to be crash-free.[2]

The system is asynchronous: the time it takes for a process to take a step and for a message to be received is unbounded. The system is *fair* however: given enough time a crash-free process takes more steps and if a message is repeatedly sent by a crash-free process to another crash-free process, then the message will eventually be received.

A consensus protocol enables processes to propose a value and guarantees that a single proposal is eventually chosen [17]. The protocol can be instantiated multiple times. The interface consists of a call propose(i, v) and an upcall decision(i, v') where i identifies the consensus instance, also known as slot number, and where v, v' are two values. Given a slot number i, we say that a process proposes v when it calls propose(i, v). A process decides on v' when it receives an upcall decision(i, v'). For any slot number i, consensus guarantees the following properties: (i) if a process decides on v, then some process proposed v (*consensus-validity*) and (ii) all processes that decide, decide the same value (*consensus-agreement*).

Although not formally proven in the Nuprl system, the synthesized consensus protocols also guarantee *consensus-termination*: if a crash-free process proposes a value, then all crash-free processes eventually decide. The FLP result says that no consensus protocol can guarantee consensus-termination in a purely asynchronous system with crash failures [18]. Thus, we need to make additional assumptions. For Paxos, we assume 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. We assume that there are infinitely many good periods for each instance of consensus to decide on a value. For 2/3-consensus, we require messages to be eventually received in the same order. Proving consensus-termination under these conditions is future work.

IV. The Broadcast Service

Our ordered broadcast service, *Aneris*, is defined by calls broadcast(v), which allows a client to broadcast message v, and deliver(s, v'), which informs a client that v' is the message delivered in slot s, where $s \geq 1$. Aneris ensures the following properties: (i) a client delivers message v in some slot s only

[2]The more common name for "crash-free" is "correct". In this paper, we use the term correct to signify that an implementation satisfies its specification.

```
process Aneris (paxos−procs, 2/3−procs, clients)
  var proposals := [];  var slot   := 1;
  var decisions := ∅;   var active := false;
  var protocol  := ‘‘2/3‘‘;

  function on_decision ((s,v))
    if not(∃s: s <= slot ∧ (s,v) ∈ decisions) then
      decisions := decisions ∪ {(s,v)};
    end if
    proposals := proposals \ {v};
    while ∃v': (slot,v') ∈ decisions do
      active := false;  perform(v');
    end while
    if not(null(proposals)) then
      propose(hd(proposals));
    end if
  end function

  function perform (v)
    if v == inl(cid,new_protocol) then
      protocol := new_protocol;
    end if
    ∀k ∈ clients: send(k,⟨‘‘deliver‘‘,(slot,v)⟩);
    slot := slot + 1;
  end function

  function propose (v)
    if not(∃s: (s,v) ∈ decisions) then
      if not(v ∈ proposals) then
        proposals := proposals @ [v];
      end if
      if not active then
        active := true;
        switch protocol
          case ‘‘paxos‘‘ : ∀loc ∈ paxos−procs:
              send(loc,⟨‘‘paxos propose‘‘,(slot,v)⟩);
          case ‘‘2/3‘‘   : ∀loc ∈ 2/3−procs:
              send(loc,⟨‘‘2/3 propose‘‘,(slot,v)⟩);
        end switch
      end if
    end if
  end function

  for ever
    switch receive()
      case ⟨‘‘broadcast‘‘,v⟩    : propose(inr v);
      case ⟨‘‘swap‘‘,v⟩         : propose(inl v);
      case ⟨‘‘decision‘‘,(s,v)⟩ : on_decision(s,v);
    end switch
  end for
end process
```

Fig. 1: The pseudo-code of our ordered broadcast service.

if some client called broadcast(v) (*broadcast-validity*), (ii) a client delivers a message at most once (*broadcast-uniqueness*), (iii) for any slot s, clients that deliver a message in slot s deliver the same message (*broadcast-agreement*), (iv) if a client is crash-free, a call to broadcast(v) eventually results in a client delivering v for some slot s (*broadcast-termination*), (v) if a client delivers v for some slot s, then all crash-free clients eventually deliver v (*broadcast-relay*). (vi) if a replica delivers some message v in some slot $s \geq 1$ then it previously delivered some message v' in slot $s-1$ (*broadcast-gap-free*).

Aneris is replicated for fault-tolerance. When broadcasting a message, a client sends a copy of the message to each replica. Because there are multiple clients, different replicas may receive messages in different orders. The replicas uses consensus to agree on the order of messages to deliver. Aneris is able to use different consensus protocols for different slots. The replicas have to agree on which consensus protocol they use for a slot, and use the broadcast service itself in order to facilitate this agreement. One can also send a special message $\langle \texttt{swap}\ p \rangle$ to request Aneris to use consensus protocol p. When deliver($i, \langle \texttt{swap}\ p \rangle$) is invoked, it signals that from slot $i+1$ consensus protocol p is being used. Such message can be sent by a monitoring service when unusual behavior of Aneris is detected. Using standard cryptographic techniques, only the monitoring service is allowed to issue swap messages, preventing an adversary from gaining control over which consensus protocol is executed.

Aneris uses $f+1$ replicas, where f is the maximum number of tolerated failures. Fig. 1 presents pseudo-code for the replicas. While functionally similar, Paxos and 2/3-consensus use different processes and Aneris has to be aware of this. After receiving a broadcast request, function propose in Fig. 1 has to send the proposed message to the processes of the current protocol. Because the consensus protocol can change, a replica cannot propose a value in slot $s+1$ until it has learned the decision of slot s (which may be a decision to use a new consensus protocol). The active flag keeps track of whether the replica is waiting for a decision.

The service keeps track of the received proposals, the decided values for each slot number, the current slot number, and the current protocol. When a proposed message v has been decided, v is removed from the proposals queue and stored in the set of decisions. The decision v is forwarded to all clients. If the decision is a request to swap protocols, the current protocol is updated. The next message, if any are left, is then proposed for the next slot. Due to asynchrony of the system, a replica may decide on a slot s before slot $s-1$. Decisions are handled in slot order to ensure that replicas use the same consensus protocol for each slot.

A. 2/3-Consensus

The 2/3-consensus protocol consists of $3f+1$ processes. The protocol is round-based. A process that receives a proposal for a particular message and slot forwards the proposal to all processes. Each process votes on the proposal and informs the other processes of their vote. Each process waits for $2f+1$ of these votes, and if the received votes are unanimous, it is decided and a *decision* message is sent to the Aneris replicas. If not unanimous, a process selects the most frequent proposal among the received votes and re-proposes this proposal for the next round. (Ties can be broken arbitrarily.)

B. Paxos

In order to tolerate up to f failures, Paxos requires $2f+1$ instances of processes called *acceptors*. Additionally, Paxos relies on $f+1$ *leaders* to get proposals accepted—a leader and an acceptor may be hosted on the same physical machine. Like 2/3-consensus, Paxos is round-based, but the rounds are called ballots and are identified by ballot numbers. Each ballot has a unique leader. When the system goes through a bad period, there may be multiple leaders that try to get different proposals decided. Ballots are designed to ensure that this cannot happen and that multiple leaders will decide the same proposal. A Paxos ballot executes in two phases. In the first phase, the leader collects information about prior ballots and previous proposals. In the second phase the leader tries to get a majority of acceptors to vote for a proposal that is consistent with prior

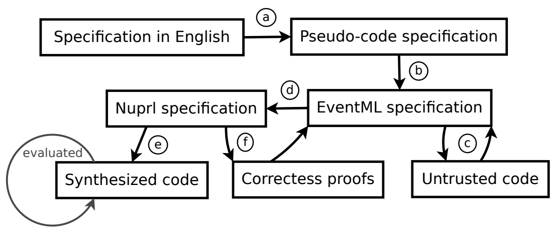

Fig. 2: Synthesis and verification workflow in Nuprl.

ballots. In particular, if a prior ballot decided a proposal, then the first phase will detect this proposal.

V. Synthesizing the Broadcast Service

A. Methodology

Fig. 2 illustrates our workflow to obtain correct-by-construction code from high-level specifications. Given an informal protocol specification, e.g., a specification in English or pseudo-code, we generate by hand a corresponding EventML specification (see markers ⓐ and ⓑ in Fig. 2). EventML is an ML dialect targeted to develop networks of distributed processes that react to *events*, where 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 design systems by specifying their information flow, and the tool automatically synthesizes code from such specifications (see marker ⓒ in Fig. 2). EventML features a collection of combinators corresponding to combinators of the Logic of Events [5], [6], a logical framework implemented in Nuprl to reason about and synthesize distributed systems. We believe these combinators provide a "workable" abstraction for structuring and reasoning about large systems because of the following computation/logic dualism: (1) each these combinators is implementable, and (2) logical formulas allow us to reason about their meaning.

The code synthesized by EventML (see marker ⓒ in Fig. 2) is not formally guaranteed to be correct because the tool has not interacted with Nuprl to generate that code yet. This untrusted code is however useful for debugging purposes. 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 a specification (see marker ⓓ in Fig. 2) which is a collection of predicates expressed in 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 use a small number of implementable combinators, it is straightforward to prove that a specification is implementable. One can then extract from such a proof the above mentioned collection of processes which is provably correct w.r.t. the specification (see marker ⓔ in Fig. 2). The synthesized code is a collection of Nuprl terms that can be evaluated using any of the term evaluators available in Nuprl and EventML. Since processes react to and produce messages, EventML features a messaging system that allows the synthesized code to run on multiple machines.

We then prove that the specified protocol satisfies correctness criteria such as the consensus properties presented in Sec. III. Our logical framework allows to do that reasoning at the specification level rather than at the code level (see marker ⓕ in Fig. 2). Finding bugs at this stage, i.e., while proving a protocol's correctness, remains costly. However, this is balanced by the fact that EventML can synthesize running code that can be used for debugging purposes.

B. Synthesis and Verification of a Broadcast Service

1) Synthesis: As illustrated in Fig. 2, one of the steps of our methodology consists in translating the informal Aneris pseudo-code into a more formal EventML specification. Fig. 3 shows the EventML translation of the "propose" function defined in Fig. 1. One can observe that the function expressed using pseudo-code and its EventML version are similar. The variable cmdt is the value to propose, it is a "tagged" command, i.e., either an inl (the left injection of a swap message) or an inr (the right injection of a broadcast message). The tuple (slot, active, proposals, decisions, protocol) is the current state of the process running propose. The propose function returns the updated state (slot, true, proposals', decisions, protocol) and msgs, a collection of messages to send to either the leaders of Paxos (ldrs), the paxos-procs in Fig. 1, or the processes of 2/3-consensus (locs), the 2/3-procs in Fig. 1.

Once an EventML specification type checks it can be loaded in Nuprl. Fig. 4 shows the Nuprl version of propose that EventML generates. The five variables `Cid`, `Op`, `eq_Cid`, `ldrs`, and `locs` are parameters of our specification: `Cid` is the type of command identifiers, `Op` is the type of commands, `eq_Cid` is an equality decider of command identifiers, and `ldrs` and `locs` are collections of leaders of paxos and processes of 2/3-consensus respectively. Once again, one can observe that the EventML and Nuprl versions of propose are similar.

Finally, given an implementable specification, Nuprl can synthesize a process generator, i.e., a function that given the location of a machine, returns the code that has to be installed at that location. Therefore, given a collection of locations, we can generate a collection of processes that have to be installed at these locations. Our Aneris specification is parametrized by reps, a collection of replica locations. It defines what a replica is and specifies that each location in reps runs a replica. The process generator synthesized by Nuprl is a total function that given a location returns a replica running at that location. To obtain the collection of processes that implement our specification, we then have to apply this function to each of the locations in reps.

2) Verification: We have formally proved, in Nuprl, that 2/3-consensus and Paxos satisfy the consensus-agreement and consensus-validity properties. These results are necessary to prove the validity and agreement properties of Aneris. Thanks to Nuprl's large library of facts and tactics that implement some patterns of reasoning in the Logic of Events [6], we

```
let propose cmdt (slot,active,proposals,decisions,protocol) =
  let proposals' = if bl-exists ((map snd decisions) ++ proposals) (same_command_tag cmdt)
                   then proposals
                   else proposals ++ [cmdt] in
  let msgs = if active & (bl-exists (map snd decisions) (same_command_tag cmdt)) then {}
             else if protocol = ``paxos`` then pax_propose'broadcast ldrs (slot,cmdt)
             else if protocol = ``2/3``   then tt_propose'broadcast  locs (slot,cmdt)
             else {} in
  ((slot,true,proposals',decisions,protocol), msgs) ;;
```

Fig. 3: The function “propose” in EventML.

```
Aneris_propose(Cid;Op;eq_Cid;ldrs;locs) ==
  λcmdt,z. let slot,active,proposals,decisions,protocol = z in
    let proposals' = if (∃zh∈map(λx.(snd(x));decisions) @ proposals. Aneris_same_command_tag() cmdt zh)_b
                     then proposals
                     else proposals @ [cmdt] fi in
    let msgs = if active ∧_b (∃zh∈map(λx.(snd(x));decisions). Aneris_same_command_tag() cmdt zh)_b then {}
               if list-deq(AtomDeq) protocol [paxos] then Aneris_pax_propose'broadcast() ldrs <slot, cmdt>
               if list-deq(AtomDeq) protocol [2/3] then Aneris_tt_propose'broadcast() locs <slot, cmdt>
               else {} fi  in
    <<slot, tt, proposals', decisions, protocol>, msgs>
```

Fig. 4: The Nuprl version of “propose”.

```
∀[Cid,Op:ValueAllType].∀[eq_Cid:EqDecider(Cid)].∀[eq_Op:EqDecider(Op)].∀[accpts,ldrs,locs,reps:bag(Id)].
∀[ldrs_uid:Id → ℤ]. ∀[flrs:ℤ]. ∀[es:EO'].
              (∀i:Id. ∀s:ℤ. ∀k:Cid. ∀c:Op.
                 if Aneris_main() outputs (Aneris_deliver'send() i <s, k, c>)
                 then Aneris_broadcast'base() observed <i, k, c>)
  supposing ((∀i:Id. ∀s:ℤ. ∀c:Cid × (Atom List) + (Id × Cid × Op).
                 if c23_main() outputs (c23_notify'send([decision]) i <s, c>)
                 then c23_propose'base([tt_propose]) observed <s, c>)
        and   (∀i:Id. ∀s:ℤ. ∀c:Cid × (Atom List) + (Id × Cid × Op).
                 if cpax_main() outputs (cpax_decision'send([decision]) i <s, c>)
                 then cpax_propose'base([pax_propose]) observed <s, c>)
        and  Aneris_message-constraint-p1(es))
```

Fig. 5: The request-validity property as stated in Nuprl.

were able to prove broadcast-validity in a matter of a few hours, and we anticipate being able to prove the other validity and agreement properties in a matter of a few days. Fig. 5 shows the statement we have proved (where we make explicit that we need consensus-validity for both 2/3-consensus and Paxos). The formula states that if at event `e`, Aneris (named `Aneris_main()` in Nuprl) sends a deliver message to location `i` containing the triple (slot number/command identifier/command) `<s, k, c>` then there exists an earlier event `e'` at which the pair `<k, c>` was proposed (i.e., Aneris received a broadcast message). It was easily proved by tracing back the information flow of the system, i.e., tracing back the outputs of Aneris to the state variables of its processes and their inputs. To prove that Aneris's safety properties hold, we also need to assume that decisions received by Aneris are sent by either 2/3-consensus or Paxos, and that the proposals received by 2/3-consensus and Paxos are sent by Aneris. Proposition `Aneris_message-constraint-p1(es)` in Fig. 5 is a Nuprl definition that states these various necessary assumptions.

VI. Preliminary Performance Results

We currently have a working prototype of Aneris running on a cluster of Linux machines connected by a gigabit switch. Aneris runs on 4 quad-core 3.6 Ghz Intel Xeons with 4GB of RAM each. We set $f = 1$, 2/3-consensus is thus using 4 machines, while Paxos only 3. The synthesized code runs inside our SML evaluator.

For testing, we ran a client on a dual-core 2.8 Ghz AMD Opteron with 4GB of RAM. We ran experiments to measure the average time it took for the client to broadcast a message and deliver it, using either Paxos or 2/3-consensus. We obtained 1.9 seconds and 2.5 seconds for Paxos and 2/3-consensus respectively.

These performance numbers are currently prohibitive for most applications. We are currently working on identifying major bottlenecks in our synthesized code and term evaluators, and are applying compilation optimizations techniques to the code synthesizer.

VII. Conclusion

We presented a correct-by-construction ordered broadcast service that allows inputs to an application to be delivered in the same order at the application replicas. Our service offers diversity in time, by allowing to dynamically change the consensus protocol, and in space, by offering the possibility to run the synthesized code in an SML or OCaml evaluator.

One could argue that our approach does not result in more reliable code as there might be bugs in the code synthesizer and evaluators. Although we recognize that this could be an issue, Nuprl is based on the LCF tactic mechanisms [19], 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.

As future work, we intend to make Byzantine-resilient consensus protocols available to the service—2/3-consensus can be quite easily adapted to tolerate such failures. We also wish to diversify the synthesized code to make process failures more independent.

Acknowledgements

This work has been partially funded by NSF, DARPA CRASH (FA8750-10-2-0238) and AFRL (FA8750-08-2-0153).

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] J. Baker, C. Bond, J. Corbett, J. J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR'11*, pages 223–234, 2011.

[4] Y. Bertot and P. Casteran. *Interactive Theorem Proving and Program Development*. Springer Verlag, 2004.

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

[6] 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.

[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] M. Bickford, C. Kreitz, and R. van Renesse. Formally verifying hybrid protocols with the NUPRL logical programming environment. Technical Report TR2001-1839, Cornell University, Ithaca, New York, 2001.

[9] M. Bickford, C. Kreitz, R. van Renesse, and X. Liu. Proving hybrid protocols correct. In R. Boulton and P. Jackson, editors, *Proceedings of 14th Int'l Conf. on Theorem Proving in Higher Order Logics (TPHOLs'01)*, volume 2152 of *Lecture Notes in Computer Science*, pages 105–120. Springer-Verlag, 2001.

[10] G. S. Blair, L. Blair, V. Issarny, P. Tuma, and A. Zarras. The role of software architecture in constraining adaptation in component-based middleware platforms. In *Middleware'00*, pages 164–184, Secaucus, NJ, USA, 2000.

[11] M. Burrows. The Chubby Lock Service for loosely-coupled distributed systems. In *SOSP'06*, Seattle, WA, 2006.

[12] Wen-Ke C., M.A. Hiltunen, and R.D. Schlichting. Constructing adaptive software in distributed systems. In *21st Int'l Conf. on Distributed Computing Systems.*, pages 635 –643, 2001.

[13] B. Charron-Bost and A. Schiper. The Heard-Of model: computing in distributed systems with benign failures. *Distributed Computing*, 22(1):49–71, 2009.

[14] L. Chen and A. Avizienis. N-version programming: A fault-tolerance approach to reliability of software operation. In *FTCS'77*, Los Alamitos, CA, 1977. IEEE Computer Society Press.

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

[16] 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.

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

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

[19] 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.

[20] J. O. Hallstrom, W. M. Leal, and A. Arora. Scalable evolution of highly available systems. *Transactions of the Institute for Electronics, Information and Communication Engineers*, pages 2154–2164, 2003.

[21] R. Hayton, A. Herbert, and D. Donaldson. Flexinet-a flexible component oriented middleware system. In *Proceedings of the 8th ACM SIGOPS European workshop on Support for composing distributed applications*, pages 17–24, New York, NY, USA, 1998. ACM.

[22] M. Isard. Autopilot: Automatic data center management. *Operating Systems Review*, 41(2):60–67, 2007.

[23] S. Jajodia, A. K. Ghosh, V. Swarup, C. Wang, and X. S. Wang, editors. *Moving Target Defense - Creating Asymmetric Uncertainty for Cyber Threats*, volume 54 of *Advances in Information Security*. Springer, 2011.

[24] J.C. Knight and N.G. Leveson. An experimental evaluation of the assumption of independence in multiversion programming. 12(1), 1986.

[25] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *CACM*, 21(7):558–565, July 1978.

[26] L. Lamport. The part-time parliament. *Trans. on Computer Systems*, 16(2):133–169, 1998.

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

[28] L. Lamport. Byzantizing paxos by refinement. In David Peleg, editor, *DISC*, volume 6950 of *LNCS*, pages 211–224. Springer, 2011.

[29] L. Lamport. Mechanically checked safety proof of a byzantine paxos algorithm, 2011.

[30] X. Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *POPL'06*, pages 42–54. ACM, 2006.

[31] D. Pavlovic, P. Pepper, and D. R. Smith. Formal derivation of concurrent garbage collectors. In Claude Bolduc, Jules Desharnais, and Béchir Ktari, editors, *MPC*, volume 6120 of *LNCS*, pages 353–376. Springer, 2010.

[32] 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 *LNCS*, pages 267–286. Springer, 2002.

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

[34] O. Rutti and A. Schiper. A predicate-based approach to dynamic protocol update in group communication. In *IPDPS'08*, pages 1–12, 2008.

[35] F.B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[36] N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan, and J. Yang. Secure distributed programming with value-dependent types. In *16th ACM SIGPLAN Int'l Conf. on Functional Programming*, pages 266–278. ACM, 2011.

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

[38] R. van Renesse, Kenneth P. Birman, Mark Hayden, Alexey Vaysburd, and David Karr. Building adaptive systems using Ensemble. *Software—Practice and Experience*, 1998.

[39] S. Weil, S.A. Brandt, E.L. Miller, D.D.E. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *OSDI'06*, 2006.