AFRL-RI-RS-TR-2013-077

FORMALLY GENERATING ADAPTIVE SECURITY PROTOCOLS

CORNELL UNIVERSITY

MARCH 2013

FINAL TECHNICAL REPORT

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

AFRL-RI-RS-TR-2013-077 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/ S /

PATRICK M. HURLEY
Work Unit Manager

/ S /

WARREN H. DEBANY, JR
Technical Advisor
Information Exploitation
and Operations Division
Information Directorate

REPORT DOCUMENTATION PAGE	*Form Approved* **OMB No. 0704-0188**

1. REPORT DATE *(DD-MM-YYYY)* MAR 2013	**2. REPORT TYPE** FINAL TECHNICAL REPORT	**3. DATES COVERED** *(From - To)* May 2008 - Oct 2012

4. TITLE AND SUBTITLE FORMALLY GENERATING ADAPTIVE SECURITY PROTOCOLS	**5a. CONTRACT NUMBER** FA8750-08-2-0153
	5b. GRANT NUMBER N/A
	5c. PROGRAM ELEMENT NUMBER 61102F & 62702F
6. AUTHOR(S) Robert L. Constable	**5d. PROJECT NUMBER** 4519
	5e. TASK NUMBER PH
	5f. WORK UNIT NUMBER 01
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Cornell University, 4130 Upson Hall, Ithaca, NY 14853; ATC- NY, Cornell Business & Technology Park, 33 Thornwood Drive, Suite 500, Ithaca, NY 14850	**8. PERFORMING ORGANIZATION REPORT NUMBER** N/A
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Laboratory/RIGA, 525 Brooks Road, Rome NY 13441-4505	**10. SPONSOR/MONITOR'S ACRONYM(S)** N/A
	11. SPONSORING/MONITORING AGENCY REPORT NUMBER AFRL-RI-RS-TR-2013-077

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
Over the five years of the project Formally Generating Adaptive Security Protocols, new formal tools based on original theoretical results of the research team allowed them to formally specify requirements for distributed protocols in a formal logic of system events. The project developed a constructive protocol description language and tools to automatically synthesize executable code from such descriptions and prove that the synthesized code satisfied formal requirements. These new automated tools produce many provably equivalent variants of the specified protocols that meet the requirements. The variants are used to change protocols on the fly to functionally equivalent protocols that have different behaviors. This ability to correctly change code on the fly creates a moving target defense that renders system more resistant to attack. This synthetic diversity defense was tested against attack scenarios that caused processes to fail, and the test system showed it was able to survive these attack scenarios.

15. SUBJECT TERMS
diversity, synthesis, verification, distributed systems, event logic, formal methods, attack resistant

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** PATRICK M. HURLEY
a. REPORT U	**b. ABSTRACT** U	**c. THIS PAGE** U	SAR	25	**19b. TELEPONE NUMBER (*Include area code)*** N/A

Contents

List of Figures

1 SUMMARY

As a result of Air Force Research Laboratory (AFRL) - Rome, Information Directorate funding and Cornell University contributions to the project *Formally Generating Adaptive Security Protocols*, the investigators from Cornell, Robert Constable, Robbert van Renesse, and Vincent Rahli, and from Architecture Technology Corporation - New York (ATC-NY), Mark Bickford, are now able to provide attack resistant capabilities to an adaptive distributed system of the kind imagined in our proposal. In particular, component protocols from the AFRL project are used in the ShadowDB distributed system. These core components are protocols synthesized automatically from high level programmable specifications. ShadowDB is an independent systems effort led by Robbert van Renesse, Nicolas Schiper, and Vincent Rahli to which we contributed attack resistant capabilities. ShadowDB can respond to cyber attack by changing on-the-fly its provably correct protocols into formally equivalent variants which are sufficiently different that they are able to resist various cyber attacks. We call such systems *attack-resistant through synthetic diversity.*

Over the five years of this research effort we created several new formal methods tools based on original theoretical results that enabled us to formally specify distributed protocols at a very high level of abstraction and automatically synthesize executable code from constructive protocol specifications. These new automated tools translate constructive specifications into executable code, and they produce many provably equivalent variants of the specified protocols. We developed a translation of synthesized code into Java and tested it against attack scenarios that caused processes to fail.

The technology we created was sufficiently robust by 2010 that we could see how to make it the basis for attack tolerance in cloud based distributed systems and in components of operating systems. These preliminary results helped us secure Defense Advanced Research Projects Agency (DARPA) funding which has allowed us to scale up the AFRL funded work and deploy a distributed database system which we call ShadowDB. Elements of that deployment and testing were also funded by this AFRL project.

We believe that our AFRL funded five year initial effort validates both the vision for synthetic diversity from the AFRL-Rome, Cornell, and ATC-NY as well as the specific technical approach proposed by Cornell and ATC-NY and the technical capabilities of the Cornell and ATC-NY research partnership in accomplishing a highly innovative and complex research task in a mode of close cooperation with AFRL project managers.

Our results demonstrate the feasibility of using formal tools to build reliably adaptive security mechanisms to protect a replicated database of the kind widely used in the industry and in the Department of Defense (DoD). We believe that our results can become a major component of the DoD's response to cyber threats as we continue to perfect our new formal methods, programming tools, and new proof technologies.

2 INTRODUCTION

We were able to support an adaptive system of the kind described in our initial proposal. It uses correct-by-construction protocols and synthetic diversity to render a distributed system more resistant to cyber attack. Our current test bed system, ShadowDB [1], is a replicated cloud database that uses our synthesized and verified consensus protocols (Multi-Paxos and 2/3-consensus) and our ability to modify them on-the-fly to respond to attacks that disable some of the processes that constitute the system. Our experiments demonstrate the feasibility of using formal methods to protect a replicated database of the kind widely used in industry and the DoD by adding diversity based adaptive security mechanisms.

We believe that our AFRL-Rome funded five year initial effort validates both the vision for synthetic diversity from the Air Force, Cornell, and ATC-NY as well as the specific technical approach proposed by Cornell and ATC-NY and the technical capabilities of the Cornell and ATC-NY research partnership in accomplishing a highly innovative and complex research task. Over the five years of this research effort we met regularly with Patrick Hurley and Kevin Kwiat of AFRL to discuss progress and ideas for moving forward. These discussions were very detailed and led to new insights and approaches. We are preparing a joint publication with AFRL collaborators Hurley and Kwiat that will relate these AFRL supported initial steps to the current state of the technology as deployed in the ShadowDB system.

The ideas we explored with the Air Force have been elaborated with additional funding from DARPA and Cornell discretionary into a broad response to the increasing threat of cyber attack and cyber warfare. We have demonstrated how to apply advanced formal methods, novel programming language technology, and proof technology to make systems attack-tolerant. Our current conception of attack-tolerance has matured with the additional DARPA funding, and we believe that our approach can evolve into a major component of the DoD's response to cyber threats as we continue to perfect our new formal methods, programming tools, and new proof technologies.

To create a system such as ShadowDB, we developed a methodology that begins with formal specification and results in provably correct and adaptable code. The formal specifications are given using the *Logic of Events* developed by Bickford and Constable in a series of articles [2, 3, 4] and related articles [5, 6, 7]. The key tool in this methodology is the Nuprl proof assistant which synthesizes many formally correct high level versions of protocol code. The proof assistant creates not only executable code, but it allows the programmer to work at a high level of abstraction with meaningful feedback in the form of the Inductive Logical Forms (ILFs). These ILFs are logical descriptions of the protocol in completely declarative language. When the ILF is proven, then we can use it to formally prove that the protocol meets its specification and has other essential properties. This guarantees properties of the synthesized code and their export to high level programs in forms that can be executed in standard programming languages.[1] We are currently able to synthesize code in Java and Lisp and are targeting other standard programming languages as well.

Over the course of the project we progressively increased the complexity of protocols we could synthesize. In 2009 we started with 2/3 consensus. In 2010 as our tools improved we synthesized a proven version of leader election and 2/3 consensus. In 2011 we did the same for a simple version of Paxos. At each stage of the evolution as our tools improved we significantly reduced the person hours involved. What took us two months in 2011 now can be done in two weeks.

We have also been able to create several proven correct code variants for the protocols. Using our methodology there are three steps where we can easily introduce diversity. This results in the ability to create 27 interchangeable variants with relative ease. We have identified additional opportunities to add proven variants and expect to be able to produce over a hundred variants of

[1]With DARPA support we have also created a specialized programming language interface to Nuprl called EventML [8]. This is a conventional ML style programming language with features designed to facilitate rapid high level coding of distributed protocols.

key protocols and eventually another order of magnitude more.

3 METHODS, ASSUMPTIONS, AND PROCEDURES

3.1 Methods

Our proposal to AFRL outlined methods that had proven themselves in previous DARPA funding of joint research between the formal methods group and the distributed systems group at Cornell [9, 10]. We also based our methods on other work between the two groups [11, 12, 13, 14, 15, 16]. Our plan was to examine the key protocols for modern distributed systems and find those which were both highly critical and amenable to formal verification and to formal synthesis. We decided to focus on consensus protocols since these are central to cloud based distributed systems and very difficult to implement and modify correctly. A great deal is known about these protocols including important theoretical results such as the Fischer/Lynch/Paterson Theorem (FLP) [17] that guides thinking about consensus. This theorem says that in the presence of possible faults, it is not possible for a group of asynchronously interacting protocols to achieve consensus. One theoretical result of our work was to show that the FLP theorem can be established constructively when consensus protocols are shown to be effectively non-blocking. This constructive version has implications for denial of service attacks on verified protocols. Our result shows that if an attacker knows a constructive proof of non-blocking for the protocol being used and controls the network, as in a data center, then that attacker can launch an undefeatable denial of service attack [18]. In this circumstance, the only defense is to be able to change the protocol. Our research has now shown how to accomplish that.

We believed that in the course of a five year research program we could start by synthesizing simple consensus protocols and by the end of the project reach the point of synthesizing the most widely used consensus protocol, Multi-Paxos [19, 20, 21]. Multi-Paxos is used by Amazon, Google, IBM, Microsoft, and a host of other companies whose systems rely on distributed computing. It is a mathematically subtle protocol that is hard to implement correctly and because of that, those who build or maintain the system are very reluctant to modify the protocol. This makes it an attractive target for attack and a point of high vulnerability in cloud based systems, especially large data warehouses and other cloud services.

The other promising protocol we considered is *virtual synchrony* [22]. We decided to start with a simpler consensus protocol which we call 2/3 consensus. This protocol can easily be made Byzantine fault-tolerant in addition to being simply fault-tolerant. By the end of the project we were indeed able to synthesize Multi-Paxos in several versions and use it together with 2/3 to make ShadowDB attack resistant. In the DARPA project we are continuing our work on Paxos in order to make it still more diverse and more efficient.

The highly regarded Cornell systems group provided frequent access to leading researchers in distributed computing. We were able to discuss our investigation of Paxos with Leslie Lamport on several occasions; he is the inventor of the algorithm. During this period Lamport was attempting to prove properties of his algorithm in the programming language TLA^+ [23] that he developed at Microsoft Research. We met with him at Cornell several times to compare our approaches. He was not able to contemplate a synthesis of the algorithm nor the verification of variants. His approach is to use model checking as the primary method of gaining confidence in its implementation. We also had long discussions with Danny Dolev, another expert in consensus protocols.

3.2 Assumptions

One of our assumptions at the beginning of the project was that our Logic of Events was fully capable of framing the synthesis problem and that the methods we had used to verify other protocols such as leader election [2] and authentication [4] would suffice for Paxos as well. In preparation for the harder protocols, we had added expressive power to the logic [24].

We learned that our assumption was wrong in the course of attempting to synthesize the core method of Paxos (called Paxos Synod). We were not able to synthesize the core of Paxos in a timely manner using the existing primitives from the Logic of Events. Our draft article from that period shows our progress toward this goal and how difficult it became to synthesize understandable code from the proofs.

In the end we judged that it would be more cost effective to increase the expressive power of the logic and start with a more abstract specification of the algorithm that could capture features used by van Renesse in his new implementation of Multi-Paxos [25] being used in other deployed systems. Another key step was the addition of a new operator, called *delegation*, to the Logic of Events [7]. An operator similar to delegation is used in the Orc system as well [26, 27], and the concept is used in the Orleans system [28]. Extending the logic in this way was entirely feasible because our base logic (Constructive Type Theory) is extremely expressive.

3.3 Procedures

Our method of operation has been to talk in depth with the systems group to understand the issues they face in building a distributed system they intend to deploy and support for a significant period. Our proposal identified a class of systems of mutual interest; so this was an easy step. We talked about the issues until we had a shared vocabulary and a shared intuitive understanding of the key protocols. Then we explored ways of formally expressing in the Logic of Events the essential properties that the system must satisfy. Since this logic was developed precisely to capture the way that our colleagues express properties of their systems, using events and message sequence diagrams, we did not face the challenge of having to explain the logic to them. This would be a problem if we had adopted temporal logics or process algebras, but from the beginning, we have used formalisms that match the way the systems group "speaks."

Once we all understood an algorithm and its requirements, we outlined how those requirements can be proved and why they could be realized in code. This sort of procedure reveals the connections between proof methods and the algorithm and allows us to write a very high level constructive specification of the algorithm using event classes and operations on them. We know by one of our theoretical results that this description is *programmable*. The next task is to prove the required properties from the full specification. This is the first iteration of our method.

The next step is to formalize system requirements in the Logic of Events and attempt to formalize our arguments that the specifications will allow us to prove the requirements. This is a very time consuming and difficult part of the task. It involves proving many results and building tactics to help automate the formal proof process. In some cases we can write proof tactics that do a significant amount of the theorem proving automatically, and it is worth letting them execute for several hours because the chances are high that they will finish most of the proof. We tend to call these proof methods “supertactics” because they automate so much of the formal proof process. With new Defense University Research Instrumentation Program (DURIP) funding in 2012, we are now able to assemble a computing platform to significantly speed up the execution of these supertactics and other parts of the protocol synthesis and diversification process.

3.4 Formal Distributed Computing Model

As background to the technical results, we include this brief overview of our process model including key concepts for reasoning about *event structures* created from computations in this model. A *system* consists of a set of *components*. Each component has a *location*, an *internal* part, and an *external* part. There may be more than one component with the same location. The internal part of a component is a *process*—its program and internal (possibly hidden) state. The external part of a component is its interface with the rest of the system. Here the interface is a list of *messages*, containing either *data* or processes and labeled with the location of the recipient. The “higher order” ability to send a message containing a process allows such systems to grow by “forking” or “bootstrapping” new components. A system executes as follows. At each step, the *environment* may choose and remove a message from the external component. If the chosen message is addressed to a location that is not yet in the system, then a new component is created at that location, using a given *boot-process*, and an empty external part.

Each component at the recipient location receives the message as input and computes a pair that contains a process and an external part. The process becomes the next internal part of the component, and the external part is appended to the current external part of the component. A potentially infinite sequence of steps, starting from a given system and using a given boot-process, is a *run* of that system. From a run of a system we derive an abstraction of its behavior by focusing on the *events* in the run. The events are the pairs, $\langle x, n\rangle$, of a location and a step at which location x gets an input message at step n, i.e. *information is transferred*. Every event has a location, and there is a natural *causal-ordering* on the set of events, the ordering first considered by Lamport [29]. This allows us to define an *event-ordering*, a structure, $\langle E,\ loc,\ <,\ info\rangle$, in which the causal ordering $<$ is transitive relation on E that is well-founded, and locally-finite (each event has only finitely many predecessors).[2] Also, the events at a given location are totally ordered by $<$. The information, $info(e)$, associated with event e is the message input to $loc(e)$ when the event occurred. We have found that requirements for distributed systems can be expressed as (higher-order) logical propositions about event-orderings.

3.4.1 Event Classes and Event Class Combinators.

A central concept in the Logic of Events is the one of *event classes*, also called event observers. Event classes observe how distributed systems agents “react” on receipt of messages. Formally, an event class of type T is a function that takes an event ordering and an event in that event ordering, and returns a bag (multiset) of elements of type T. If the class X associates the bag $\{v_1, ..., v_n\}$ with the event e, we say that X observes the values v_i’s at e.

To connect the concept of event class with the process model, we say that a class X is *programmable* if there exists a set of processes that produce the same observations as class X, i.e. the

[2]These event structures and orderings are similar in spirit to Winskel’s [30].

union of the external parts generated by the process in response to the input message associated with an event is the same as the bag of values "observed by" class X on that event.

In response to an input message (of type Msg), an event class M of type Loc $*$ Msg generates a (possibly empty) bag of observations that are pairs of a location and a message. If such an event class, which we call a main class, is programmable then we call it a *message automaton*. A message automaton M can be run as a distributed program as follows: The program implementing M is a set of processes. When a process has a message in its in-box, it computes in response the bag of location-message pairs specified by class M and these responses are added to its out-box. A message-passing layer moves messages from out-boxes to the addressed in-boxes.

We construct complex event classes from simple building blocks using *event class combinators*. For example, the combinator X || Y forms the parallel composition of classes X and Y. On each event, it observes the union of the observations of classes X and Y. An expression like F o X is a function composition, applying the function F to the information observed by class X.

Event classes have a logical aspect. Given an event e in some event structure, a class X of type T, and an element v in T, we write $v \in X(e)$ if X observes v at event e, i.e., v is in the bag of observations that the class X makes at event e. This relation between observed elements, events, and classes is called the *class relation*.

Event class combinators can be seen as having two facets: a logical one and a programming one. Each event class combinator defines the class relation for the derived class in terms of the class relation of component classes. Each event class combinator builds the program for the derived class from the programs for the component classes.

We have defined a language EventML that extends the ML language with event classes and event class combinators. EventML builds event classes from simple base classes using only event class combinators that have been proved in Nuprl to preserve programmability. The Nuprl proof assistant can then automatically prove that the main class of an EventML specification is programmable. The proof assistant can then extract the implementation (the set of processes that implement the Message Automaton) from the proof.

We realized that our systems engineering colleagues at Cornell understand and build complex distributed systems using many interacting actors, agents, or threads [25]. We were also familiar with the elegance of the *Orc* approach to protocols [26, 27] and of the architecture of the Orleans system [28]. Not all of these agents or threads are present in the initial state of the system and many are created to perform a single task and then disappear. We discovered that we can specify this kind of system architecture with event classes constructed with a *delegation combinator* (X $\rangle$= Y). This combinator turns out to be the "bind" operator that makes event classes form a monad. The class (X $\rangle$= Y) has the effect of spawning a subprocess (Y v) whenever class X observes a value v. We typically use it to decompose a complex algorithm into subtasks that are easier to define and reason about. One typical use is to define subprocess (Y v) to be a handler that sends some messages related to parameter v, gathers the responses to those messages, reports an answer (by sending a message), and then halts.

As mentioned earlier, the class relation of classes built with combinators is expressed in terms of the class relations of their components. Because of this, and the fact that all classes specified in EventML are programmable, we say that EventML is a *compositional, constructive specification language*.

As an interesting example, the class relation $v \in (X \texttt{>>=} Y)(e)$ is equivalent to

$$\exists e' \leq e.\, \textbf{loc}(e') = \textbf{loc}(e) \wedge \exists x.\, x \in X(e') \wedge y \in (Y\ x)(e)$$

Thus, as a logical operator, (X $\rangle$= Y) is akin to a "past" operator in temporal logic. A subtle point, not apparent in our notation but present in the formal *Nuprl* version, is that each class relation has an additional parameter that gives the event ordering from which the event is taken. In the formula for $v \in (X \texttt{>>=} Y)(e)$ where e is taken from event ordering eo, the expression $y \in (Y\ x)(e)$ is interpreted in the event ordering $eo.e'$ that "starts at" event e'. This accounts for the fact that $Y\ x$ was spawned at event e' and does not include earlier events in its history.

For each of the event class combinators C allowed in EventML we have proved in Nuprl an equivalence relation that expresses when an element v is observed by C. We use these lemmas to prove properties of the protocols specified in EventML.

3.4.2 Inductive Logical Forms.

The inductive logical form of a specification is a first order formula that characterizes completely the observations (the responses) made by the main class of the specification. The formula is inductive because it typically characterizes the responses at event e in terms of observations made by a sub-component at a prior event $e' < e$. Such inductive logical forms are automatically generated in Nuprl from event class definitions, and simplified using various re-writings. With an inductive logical form we can easily prove invariants of the specification by induction on causal order. We give an example of one of these forms in [31].

4 RESULTS AND DISCUSSION

The results we mentioned in the summary constitute our main accomplishments, the deployment of the system ShadowDB that uses synthesized correct by construction protocols, including 2/3 and Multi-Paxos, that are attack resistant. However, behind those accomplishments is a long list of contributions that we are able to use in our DARPA effort, that we can publish, that we teach, and that have considerably advanced the reach of constructive formal methods. We are now in a much better position to tackle even more complex tasks and support even larger systems.

We are also in a position to understand what new elements of proof technology and distributed system theory will make it possible to extend our results to large systems and to formally express other informal properties of systems that designers care about but cannot formulate precisely as of yet.

In this section we will examine in more detail one of the technical results that illustrates several of the points we have discussed above. We will examine a version of the 2/3 consensus protocol that is written in EventML code. We show one of the variants of the code that can be created from the very high level constructive specification in the Logic of Events. That specification uses event combinators [24] and other highly original results of this research such as the delegation operator [7] and recursive delegation operator that require considerable explanation as provided in the cited articles and new publications being prepared [24, 7, 1, 8].

4.1 Two-Thirds Consensus

We present here code for 2/3-consensus in the style of van Renesse's algorithm for Multi-Paxos [25]. This code corresponds to what is generated from the high level event combinators using delegation, and it shows the kind of concrete code that can be produced from the more abstract event combinators. Readers can get a sense of the kind of diversity that results from the synthesis process. We will point out a variant of this concrete code that can result from a change of the abstract event combinators during synthesis. We show this after presenting the concrete 2/3-consensus code.

Other variants at this level of concrete system code can be generated by changing the event combinators slightly. For example, we could use the recursive delegation combinator and generate a different communication pattern in the concrete code. From the level of concrete code we display here, which most distributed systems programmers will recognize, it is easy to see that we can generate more diversity by changing the concrete data structures.

Also note that this protocol builds a distributed state machine that handles the actual processing in systems like ShadowDB. In ShadowDB we also achieve diversity by switching between 2/3 and Paxos. There are several opportunities to diversify Paxos that we do not discuss here.

This 2/3-consensus service responds to commands issued to any of the replicas, which must come to consensus on the order in which those commands are to be performed, so that all replicas process commands in the same order. Replicas may crash. This 2/3-consensus protocol is a simple protocol for coming to consensus in a manner that tolerates f failures by using (precisely) $3f + 1$ replicas. As opposed to Paxos, in this protocol there is no leader driving the decisions. All the replicas are driving the decisions concurrently by voting for commands, and broadcasting their votes to the other replicas. 2/3-consensus is *round based*, meaning that if no quorum of $2f + 1$ replicas can reach consensus after one exchange of votes, then they will try again at a higher round, and this continues forever until consensus is reached. Each round requires a single phase, meaning that each replica only has to send one message per round to each of the other replicas. Note that a single phase consensus protocol that uses only $2f + 1$ replicas cannot solve consensus. 2/3-consensus achieves consensus using $3f + 1$ replicas. In the next few paragraphs we present this protocol in more detail by breaking it down into three small interacting processes. These are processes represented at the abstract level of event combinators such as delegations.

As shown in Figure 1, replicas receive proposals of the form slot number/command (messages with header "$\mathsf{propose}$"). Given a proposal $(\mathsf{slot}, \mathsf{cmd})$, if no command has already been proposed for slot number slot (condition captured by the formula $\mathsf{slot} > \mathsf{max} \vee \mathsf{slot} \in \mathsf{missing}$) then the replica updates its state and spawns off a voter to handle that proposal, i.e., to reach consensus on which command to associate to slot number slot.

As shown in Figure 2, a voter works on a single slot number, say slot. It is in charge of reaching consensus with the other replicas on which command to associate with slot. As mentioned above, 2/3-consensus is round based. Each voter initially starts voting at round 0. If consensus is reached then the voter notifies the clients and exits. Consensus is reached when a voter receives a message with header "$\mathsf{decided}$" from one of its sub-processes. Messages with header "retry" are sent by sub-processes to notify the voters that consensus has not been reached at a given round (specified in the body of the message). Note that a voter can also start a new round when it receives a vote for a round higher than its current round, which can happen, e.g., if that voter missed some rounds because it became slow. Without that the protocol would not be "live".

```
process Replica (replicas, clients, failures)
  var proposals := ∅;

  function propose (slot,cmd)
    if slot ∉ proposals then
      proposals := {slot} ∪ proposals;
      spawn(Voter(slot,cmd, replicas, clients, failures));
    end if
  end function

  for ever
    switch receive()
      case ⟨``propose``,(slot,cmd)⟩ : propose(slot,cmd);
      end case
      case ⟨``vote``,(slot,round,cmd,loc)⟩ : propose(slot,cmd);
      end case
    end switch
  end for
end process
```

Figure 1: Replica Process

```
process Voter (slot,cmd, replicas, clients, failures)
  var round := 0;

  spawn(Round(self(), slot, round, cmd, replicas, failures));

  function vote (r,c)
    if r > round then
      round := r;
      spawn(Round(self(), slot, r, c, replicas, failures));
    end if
  end function

  for ever
    switch receive()
      case ⟨``retry``,(round,cmd)⟩ : vote(round,cmd) ;
      end case
      case ⟨``vote``,(n,round,cmd,loc)⟩ : if n = slot then vote(round,cmd); end if
      end case
      case ⟨``decided``,cmd⟩ :
        ∀ client ∈ clients : send(client,⟨``notify``,(slot,cmd)⟩);
        exit();
      end case
    end switch
  end for
end process
```

Figure 2: Voter Process

As shown in Figure 3, each round r is handled by a process called Round which is in charge of gathering a quorum of votes for a given slot number at round r. Consensus is reached if $2f + 1$ replicas vote for the same command. This is computed using the possmaj function which is left undefined here. At this point we can see a clear opportunity for diversity at this level in the implementation of this *possmaj* function since the ballots can be collected in a variety of data structures.

Another source of diversity is that we can run this algorithm with $5f + 1$ replicas and thereby tolerate f Byzantine failures.

If consensus is reached or if a new round of votes has to be started, then in both cases the round handler notifies the corresponding voter.

```
process Round (voter,slot,round,cmd,replicas,failures)
  var cmds    := ∅;
  var senders := ∅;

  ∀ loc ∈ replicas : send(loc,⟨``vote``,(slot,round,cmd,self())⟩);

  function vote (c,sender)
    if sender ∉ senders then
      cmds := {c} ∪ cmds;
      senders := {sender} ∪ senders;
      if |senders| = 2 ∗ failures + 1 then
        let (nbocc,cmd') = possmaj cmds c in
        if nbocc = 2 ∗ failures + 1 then
          send(voter,⟨``decided``,cmd'⟩);
          exit();
        else
          send(voter,⟨``retry``,(round+1,cmd')⟩);
          exit();
        end if
      end if
    end if
  end function

  for ever
    switch receive()
      case ⟨``vote``,(n,r,cmd,sender)⟩ : if n = slot ∧ r = round then vote(cmd,sender); end if
      end case
    end switch
  end for
end process
```

Figure 3: Round Process

Figure 4 illustrates a simple run of the system on one proposal sent to the first replica. It shows how replicas spawn off voters which in turn spawn off round handlers.

Figure 5 presents an instance where one round is not enough for the replicas to reach consensus. In that figure replicas R1 and R2 vote for 0 while replicas R3 and R4 vote for 1. Replicas R1 and R2 gather votes from R1, R2, and R3, and because the votes are not unanimous, consensus has not been reached. In the next round, they will vote again for 0 which is the majority. Replicas R1 and R3 gather votes from R2, R3, and R4, and again because the votes are not unanimous, consensus has not been reached. In the next round, they will vote again for 1 which is the majority. Note that this could go on forever, if at least one machine does not start voting for another value.

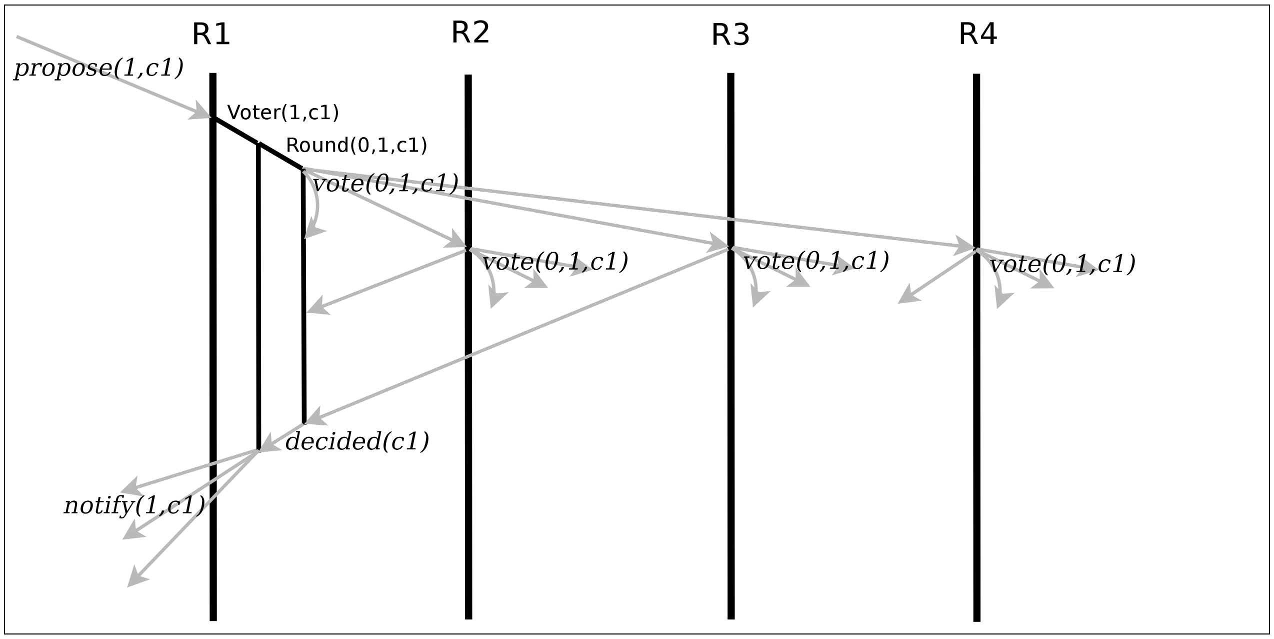

Figure 4: Spawning of sub-processes

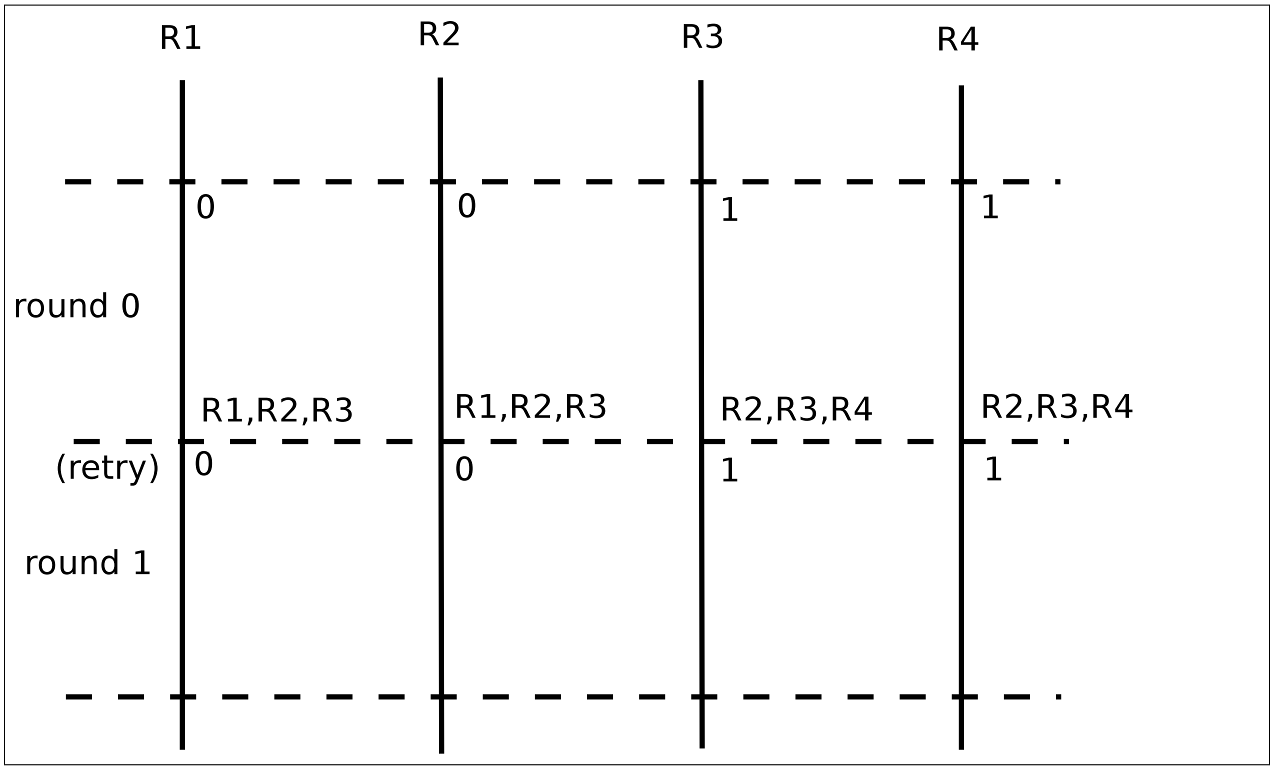

Figure 5: Retry

4.2 A Variant of the Two-Thirds Consensus Protocol

Note that each time a round process sends a “ retry “ message to a voter, that voter spawns off a new round process. Formally, this is done using our delegation combinator. Figures 6 and 7 specify a variant of 2/3-consensus that defines rounds as recursive processes instead. Formally, this is done using our recursive delegation combinator. This variant results in a different messaging pattern. In this variant, voters are simply in charge of spawning initial rounds (i.e., for round number 0). Instead of sending a “ retry “ message to its parent voter process, in this variant a round handler spawns off a new round handler for a higher round. In addition, round handlers are now also in charge of spawning off new round handlers when receiving votes for higher rounds.

```
process Voter (slot,cmd,replicas,clients,failures)
  spawn(Round(slot,0,cmd,replicas,clients,failures));
end process
```

Figure 6: Voter Process (variant)

```
process Round (slot,round,cmd,replicas,clients,failures)
  var cmds    := ∅;
  var senders := ∅;

  ∀ loc ∈ replicas : send(loc,⟨''vote'',(slot,round,cmd,self())⟩);

  function vote (c,sender)
    if sender ∉ senders then
      cmds := {c} ∪ cmds;
      senders := {sender} ∪ senders;
      if |senders| = 2 ∗ failures + 1 then
        let (nbocc,cmd') = possmaj cmds c in
        if nbocc = 2 ∗ failures + 1 then
          ∀ client ∈ clients : send(client,⟨''notify'',(slot,cmd')⟩);
          exit();
        else
          spawn(Round(slot,round+1,cmd',replicas,clients,failures));
          exit();
        end if
      end if
    end if
  end function

  for ever
    switch receive()
      case ⟨''vote'',(n,r,cmd,sender)⟩ :
        if n = slot ∧ r >= round then
          if r = round then vote(r,cmd,sender);
          else
            spawn(Round(slot,r,cmd',replicas,clients,failures));
            exit();
          end if
        end if
      end case
    end switch
  end for
end process
```

Figure 7: Round Process (variant)

4.3 Verification of Two-Thirds Consensus

The basic safety properties of any consensus protocol are *agreement* and *validity*. Both these properties have been formally proved in the Logic of Events implemented in Nuprl, for the 2/3-consensus protocol presented above. We state them in terms of notifications (i.e., messages with header " notify "). In the Logic of Events, we specify distributed programs using what we call *event classes* or *event observers*. Such an event class observes part of the information flow of a distributed system. In order to discuss our effort on verifying the safety properties of 2/3-consensus, we first need to introduce such event classes. For example base classes allow one to observe the content of input messages. Let notify 'base be the base class that observes receipt of " notify " messages, and let propose'base be the base class that observes receipt of " propose" messages. System properties are predicates on event orderings which are abstract representations of distributed system runs; we must prove that the predicates are true of all possible runs of the system consistent with our specification of 2/3-consensus.

Agreement says that notifications never contradict one another:

```
∀[Cmd:ValueAllType]. ∀[cmdeq:EqDecider(Cmd)]. ∀[locs,clients:bag(Id)].
∀[coeff:{2...}]. ∀[flrs:ℕ]. ∀[es:EO'].
  ((StandardAssumptions(Main) es)
  ⇒ bag-no-repeats(Id;locs)
  ⇒ (bag-size(locs) = ((coeff * flrs) + flrs + 1))
  ⇒ any v1,v2 from notify'base(Cmd) satisfy
     ((fst(v1)) = (fst(v2))) ⇒ ((snd(v1)) = (snd(v2))))
```

Slot numbers are represented as integers, and Cmd is the type of commands. We assume that equality on types is decidable, i.e., we assume that cmdeq is an equality decider on Cmd. The location bag locs is a collection of $(3f+1)$ replicas, and the location bag clients is a collection of clients. The natural number coeff is instantiated to 2 in 2/3-consensus. The natural number flrs is the number of tolerated failures. Finally, es is an event ordering. As mentioned above we must prove our properties on all event orderings.

The assumption (StandardAssumptions(Main) es) says among other things that messages with headers "vote", " retry ", "decided", and " notify " are sent by Main, our 2/3-specification. The assumption bag-no-repeats(Id;locs) constrains the replica collection locs to have no repeats. The assumption (bag-size(locs) = ((coeff * flrs) + flrs + 1)) constrains the replica collection to be of size $3f+1$ when coeff is 2.

Finally, we have proved that if two locations receive notifications v1=(s1,c1) and v2=(s2,c2) such that s1=s2 then c1=c2.

Validity says that any proposal decided on must be one that was proposed:

```
∀[Cmd:ValueAllType]. ∀[cmdeq:EqDecider(Cmd)]. ∀[locs,clients:bag(Id)].
∀[coeff:{2...}]. ∀[flrs:ℕ]. ∀[es:EO'].
  ((StandardAssumptions(Main) es)
  ⇒ for every d in notify'base(Cmd) there is an
     earlier  p in propose'base(Cmd) such that
     d = p)
```

As explained above, in order to ease the verification of the correctness of a distributed system, we generate Inductive Logical Forms that completely characterize the information flow of the system in a logical and declarative language.

For example the following formula expresses when votes are cast:

```
∀[Cmd:ValueAllType]. ∀[clients:bag(Id)]. ∀[cmdeq:EqDecider(Cmd)].
∀[coeff,flrs:ℤ]. ∀[locs:bag(Id)]. ∀[es:EO']. ∀[e:E].
∀[receiver:Id]. ∀[slot,rnd:ℤ]. ∀[c:Cmd]. ∀[sndr:Id].
  (<receiver, vote'msg(Cmd;<<<slot,rnd>,c>,sndr>)> ∈ Main(e)
  ⟺ loc(e) ↓∈ locs
      ∧ (receiver ↓∈ locs ∧ (sndr = loc(e)))
      ∧ (↓∃e':{e':E| e' ≤loc e }
           ((↓∃max:ℤ
               ∃missing:ℤ List
                (<max, missing> ∈ ReplicaState(Cmd)(e')
                 ∧ ((max < slot) ∨ (slot ∈ missing))))
           ∧ (∃c':Cmd
               ((↓((e = e') ∧ (c = c') ∧ (rnd = 0))
                  ∨ ((↓∃e1:{e1:E| e1 ≤loc e }
                        (((↓∃maxr:ℤ. (maxr ∈ NewRoundsState(Cmd) slot(e1)
                                     ∧ (maxr < rnd)))
                        ∧ (<<slot,rnd>,c> ∈ retry'base(Cmd)(e1)
                          ↓∨ (∃sndr':Id.
                               <<<slot,rnd>,c>,sndr'> ∈ vote'base(Cmd)(e1))))
                        ∧ (e = e1)))
                     ∧ (no Notify(Cmd;clients) slot between e' and e)))
               ∧ (<slot,c'> ∈ propose'base(Cmd)(e')
                 ↓∨ (∃rnd':ℤ. ∃sndr':Id.
                      <<<slot,rnd'>,c'>,sndr'> ∈ vote'base(Cmd)(e'))))))))
```

This formula roughly says that a replica (let us call that replica R) sends a vote of the form <<<slot,rnd>,c>,sndr> at event e to receiver iff the following holds: (1) the two locations R and receiver are members of the replica collection locs, and sndr is R. (2) There exists an event e' prior to e such that: (a) at that time the slot number slot had never been seen by R; (b) there exists a command c' which is either c if R is still at the initial round 0, or was received in a " retry " message, or was received in a " vote " message for a higher round than R's current round; (c) c' was received at e' either in a " propose" message or a " vote " message.

Using such ILFs, we can easily track down the information flow of a system from outputs to inputs and therefore we can easily prove properties such as validity properties.

4.4 Components of the Work and Independent Validation

We have explicitly mentioned in this report the components essential to this work. Other research teams would be able to validate our results. One approach would be to use our tools. We support other researchers who wish to use Nuprl by making the system and its extensive formal digital library available. We provide extensive information about the system on the Nuprl web page, www.nuprl.org. The Nuprl user manual and tutorial by Professor Christoph Kreitz have served as a good introduction for new users [32]. Our formal libraries include most of the results that were used in this effort, and we continue to add new material as the project advances. These online resources are extensive, amounting to well over two hundred pages of written material if printed.

Another interesting and plausible way to validate our results would be to take advantage of the portability of the Logic of Events. We designed this theory to be expressible in first-order logic.

Thus it can be implemented by many other proof assistants, even classical assistants in the style of HOL [33, 34, 35, 36]. This logic has been studied at other universities, and we know from results at Yale [37] that it could be used by Coq in much the same way as it is used in Nuprl.

5 CONCLUSIONS

As stated in the summary, we believe that our results can become a major component of the DoD's response to cyber threats as researchers continue to improve new formal methods of the kind we pioneered here. We believe that other researchers will experiment with formal synthesis and synthetic diversity. We think others will develop new programming tools such as EventML, and they will exploit new proof technologies such as our distributed super-tactics.

We believe that one of the most effective technologies we have created over the years to support our work is the ability to rapidly replay the entire development of a system in order to improve performance or formally establish additional properties or modifications of currently proved properties. We made extensive use of this replay technology in deploying the new formal methods tools we created during this project. We think our results will encourage wider development of replay technology.

We will be sorting through the possible next steps and seeking sources of funding to advance this line of work which we see as one of the surest ways to protect DoD software infrastructure from cyber attack. It is also one of the surest ways to enhance US technical leadership in a highly advanced technology which the United States and the European Union currently dominate.

6 REFERENCES

References

[1] Nicolas Schiper, Vincent Rahli, Robbert Van Renesse, Mark Bickford, and Robert L. Constable. ShadowDB: A replicated database on a synthesized consensus core. In *Eighth Workshop on Hot Topics in System Dependability*, HotDep, 2012.

[2] Mark Bickford and Robert L. Constable. A logic of events. Technical Report TR2003-1893, Cornell University, 2003.

[3] Mark Bickford and Robert L. Constable. A causal logic of events in formalized computational type theory. In *Logical Aspects of Secure Computer Systems, Proceedings of International Summer School Marktoberdorf 2005*, to Appear 2006. Earlier version available as Cornell University Technical Report TR2005-2010, 2005.

[4] Mark Bickford and Robert L. Constable. Formal foundations of computer security. In *Formal Logical Methods for System Security and Correctness*, volume 14, pages 29–52, 2008.

[5] Mark Bickford, Robert Constable, Joseph Y. Halpern, and Sabina Petride. Knowledge-based synthesis of distributed systems using event structures. In Franz Baader and Andreo Voronsky, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 3452 of *Lecture Notes in Computer Science*, pages 449–465, 2005.

[6] Mark Bickford, Robert L. Constable, Joseph Y. Halpern, and Sabina Petride. Knowledge-based synthesis of distributed systems using event structures. *Logical Methods in Computer Science*, 7(2), 2011.

[7] Mark Bickford, Robert Constable, and Vincent Rahli. The Logic of Events, a framework to reason about distributed systems. Computing and Information Science Technical Reports http://hdl.handle.net/1813/28695, Cornell University, Ithaca, NY, 2012.

[8] Vincent Rahli. Interfacing with proof assistants for domain specific programming using eventml. In *10th International Workshop On User Interfaces for Theorem Provers*, 2012.

[9] Xiaoming Liu, Christoph Kreitz, Robbert van Renesse, Jason J. Hickey, Mark Hayden, Kenneth Birman, and Robert Constable. Building reliable, high-performance communication systems from components. In David Kotz and John Wilkes, editors, *17th ACM Symposium on Operating Systems Principles (SOSP'99)*, volume 33(5) of *Operating Systems Review*, pages 80–92. ACM Press, December 1999.

[10] Ken Birman, Robert Constable, Mark Hayden, Jason Hickey, Christoph Kreitz, Robbert van Renesse, Ohad Rodeh, and Werner Vogels. The Horus and Ensemble projects: Accomplishments and limitations. In *DARPA Information Survivability Conference and Exposition (DISCEX 2000)*, pages 149–161, Hilton Head, SC, 2000. IEEE Computer Society Press.

[11] Kenneth P. Birman and Robbert van Renesse, editors. *The Isis Book: Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press, 1994.

[12] Robbert van Renesse, Takako Hickey, and Kenneth P. Birman. Design and performance of Horus: A lightweight group communications system. Department of Computer Science TR94-1442, Cornell University, Ithaca, NY, 1994.

[13] Xiaoming Liu, Robbert van Renesse, Mark Bickford, Christoph Kreitz, and Robert Constable. Protocol switching: Exploiting meta-properties. In Luis Rodrigues and Michel Raynal, editors, *International Workshop on Applied Reliable Group Communication (WARGC 2001)*, pages 37–42. IEEE, 2001.

[14] Mark Bickford, Christoph Kreitz, Robbert van Renesse, and Robert Constable. An experiment in formal design using meta-properties. In J. Lala, D. Mughan, C. McCollum, and B. Witten, editors, *DARPA Information Survivability Conference and Exposition II (DISCEX-II)*, volume II of *IEEE Computer Society Press*, pages 100–107, Anaheim, CA, 2001.

[15] Mark Bickford, Christoph Kreitz, Robbert van Renesse, and Xiaoming Liu. Proving hybrid protocols correct. In Richard Boulton and Paul Jackson, editors, *14th International Conference on Theorem Proving in Higher Order Logics*, volume 2152 of *Lecture Notes in Computer Science*, pages 105–120, Edinburgh, Scotland, September 2001. Springer-Verlag.

[16] L. Zhou, F. Schneider, and R. van Renesse. COCA: A secure distributed on-line certification authority. *ACM Transactions on Computer Systems*, 20(4):329–368, November 2002.

[17] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faculty process. *JACM*, 32:374–382, 1985.

[18] Robert L. Constable. Effectively nonblocking consensus procedures can execute forever: a constructive version of flp. Technical Report Tech Report 11512, Cornell University, 2008.

[19] Leslie Lamport. The part-time parliament. *ACM Trans. Computer Systems*, 16(2):133–169, 1998.

[20] R. De Prisco, B. Lampson, and N. Lynch. Revisiting the paxos algorithm. *Theoretical Computer Science*, 243:35 – 91, 2000.

[21] Leslie Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):51–58, 2001.

[22] Kenneth P. Birman and Thomas A. Joseph. Exploiting virtual synchrony in distributed systems. *Proc 11th Symposium on Operating Systems Principles (SOSP)*, pages 123–138, November 1987.

[23] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, Boston, 2003.

[24] Mark Bickford. Component specification using event classes. In *Lecture Notes in Computer Science 5582*, pages 140–155. Springer, 2009.

[25] Robbert van Renesse. Paxos made moderately complex. Technical report, Cornell University, 2011.

[26] David Kitchin, William R. Cook, and Jayadev Misra. A language for task orchestration and its semantic properties. In Christel Baier and Holger Hermanns, editors, *CONCUR*, volume 4137 of *Lecture Notes in Computer Science*, pages 477–491. Springer, 2006.

[27] David Kitchin, Adrian Quark, William R. Cook, and Jayadev Misra. The Orc programming language. pages 1–25, 2009.

[28] S. Bykov, A. Geller, G. Kliot, J. Larus, Rl Pandya, and J. Thelin. Orleans: A framework for cloud computing. Technical Report MSR-TR-2010-159, Microsoft Research, 2010.

[29] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–65, 1978.

[30] G. Winskel. *Events in Computation*. PhD thesis, University of Edinburgh, 1980.

[31] Vincent Rahli. Aneris: A diversied and correct-by-construction broadcast service., 2012. Presentation given at WRiPE2012.

[32] Christoph Kreitz. The Nuprl Proof Development System, version 5, Reference Manual and User's Guide. Cornell University, Ithaca, NY, 2002. `http://www.nuprl.org/html/02cucs-NuprlManual.pdf`.

[33] Michael Gordon and Tom Melham. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic*. Cambridge University Press, Cambridge, 1993.

[34] John Harrison. HOLLight: A tutorial introduction. In *Formal Methods in Computer-Aided Design (FMCAD'96)*, volume 1166 of *Lecture Notes in Computer Science*, pages 265–269. Springer, 1996.

[35] Amy P. Felty and Douglas J. Howe. Hybrid interactive theorem proving using NUPRL and HOL. In *CADE 97, LNAI 1249*, pages 351–365. Springer.

[36] Pavel Naumov, Mark-Olivar Stehr, and José Meseguer. The HOL/NUPRL proof translator: A practical approach to formal interoperability. In *TPHOLS 2001, LNCS 2152*, pages 329–345. Springer.

[37] Zhong Shao. Certified software. *Communications of the ACM*, 53:56–66, 2010.

LIST OF SYMBOLS, ABBREVIATIONS, AND ACRONYMS

AFRL	Air Force Research Laboratory
ATC-NY	Architecture Technology Corporation - New York
DARPA	Defense Advanced Research Projects Agency
DoD	Department of Defense
DURIP	Defense University Research Instrumentation Program
FLP	Fischer/Lynch/Paterson Theorem
ILF	inductive logical form