Investigating Correct-by-Construction Attack-Tolerant Systems

Robert Constable, Mark Bickford, Robbert van Renesse
Department of Computer Science
Cornell University
Ithaca, NY
Email: {rc,markb,rvr}@cs.cornell.edu

***Abstract*—*Attack-tolerant* distributed systems change their protocols *on-the-fly* in response to apparent attacks from the environment; they substitute functionally equivalent versions possibly more resistant to detected threats. Alternative protocols can be packaged together as a single *adaptive protocol* or variants from a *formal protocol library* can be sent to threatened groups of processes. We are experimenting with libraries of attack-tolerant protocols that are correct-by-construction and testing them in environments that simulate specified threats, including constructive versions of the famous FLP imaginary adversary against fault-tolerant consensus. We expect that all variants of tolerant protocols are automatically generated and accompanied by *machine checked proofs* that the generated code satisfies formal properties.**

I. Introduction

A. Background

Using a constructive Logic of Events based on Computational Type Theory (CTT) [ABC06], [CB08], [Bic09] we have been able to formally specify safety and liveness properties for distributed protocols and synthesize executable code from constructive proofs that the specifications are realizable [CB08]. We have used this *proofs-as-processes* method to build fault-tolerant protocols, provably secure protocols, and adaptive protocols [LKvR+99]. Recently we have created versions of the important Paxos consensus protocol [Lam01] this way. This basic methodology has led us to experiment with protocols that we think of as *attack-tolerant* in the sense that they can respond to perceived threats from the environment such as denial of service attacks, message order attacks, Byzantine attacks, and other threats. These attack-tolerant protocols will respond by adapting *on-the-fly* to alternative versions that are also known to be provably correct and provide the same functionality. This system development capability is based on a computational semantics for assertions in our standard Logic of Events. This logic is based on the concept of *event structures* [Win89], [Abr99] which are defined over executions of process in the *standard model* of asynchronous message passing computation [Lyn96], [AW04]. This semantics is expressed in CTT in such a way that proof terms contain *distributed realizers*. These realizers are abstract processes which can be compiled into appropriate programming languages such as Java, ML, Erlang, $F^{\#}$, etc. A key step in making this methodology practical is the use of *programmable event classes* to specify computing tasks at a high level of abstraction that can be refined automatically to processes. Recently we have extended our methodology to a broader notion of process, a *General Process Model* [BC10a], broad enough to encompass the higher-order π-calculus [Mil09] and other *process algebras* that include Petri nets. In the new model we can synthesize and reason about *mobile processes* as well.

B. Outline

In the next section we describe our formal computing model and the basic concepts needed to talk about mobile processes and to state assumptions on the computing environment. A simple consensus protocol illustrates the key ideas. In the final section we discuss the construction of attack-tolerant protocols based on *synthetic code diversity* and show how to state properties of the environment and create experimental computing platforms to test attack-tolerant protocols, for instance by launching a provably unbeatable attack with a *constructive* version of the Fischer/Lynch/Paterson result [FLP85].

II. Formal Distributed Computing Model

Here is a brief overview of our new General Process Model [BC10a] of distributed computation. We mention key concepts for reasoning about event structures created from these computations. A *system* consists of a set of *components*. Each component has a *location*, an *internal* part, and an *external* part. Locations are just abstract identifiers. 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. In this account, the interface will be a list of *messages*, containing either *data* or processes, labeled with the location of the recipient. The "higher order" ability to send a message containing a process allows a system 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 [Lam78]. 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). 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. To illustrate this and motivate the results in the rest of the paper we present a simple example of *consensus* in a group of processes.

Example 1. *A simple consensus protocol: TwoThirds*

Each participating component will be a member of some groups and each group has a name, G. A message $\langle G, i\rangle$ from the environment to component i informs it that it is in group G. The groups have $n = 3f+1$ members, and they are designed to tolerate f failures. When any component in a group G receives a message $\langle [start], G\rangle$ it starts the consensus protocol whose goal is to decide on values received by the members from *clients*. We assume that once the protocol starts, each process has received a value v_i or has a default non-value. The simple TwoThirds consensus protocol is this: A process P_i that has a value v_i of type T starts an *election* to choose a value of type T (with a decidable equality) from among those received by members of the group from clients. The elections are identified by natural numbers, el_i initially 0, and incremented by 1, and a Boolean variable $decide_i$ is initially $false$. The function from lists of values, Msg_i to a value is a parameter of the protocol. If the type T of values is Boolean, we can take f to be the majority function.

Begin
Until $decide_i$ **do**:
1. **Increment** el_i; 2. **Broadcast** vote $\langle el_i, v_i\rangle$ to G;
3. **Collect** in list Msg_i $2f + 1$ votes of election el_i;
4. **Choose** v_i := $f(Msg_i)$;
5. **If** Msg_i is unanimous **then** $decide_i$:= $true$
End

We describe protocols like this by classifying the events occurring during execution. In this algorithm there are *Input*, *Vote*, *Collect*, and *Decide* events. The components can recognize events in each of these *event classes* (in this example they could all have distinctive headers), and they can associate information with each event (e.g. $\langle e_i, v_i\rangle$ with Vote, Msg_i with Collect, and $f(Msg_i)$ with Decide). Events in some classes *cause* events with related information content in other classes, e.g. Collect causes a Vote event with value $f(Msg_i)$. In general, an *event class* X is a function on events in an event ordering that *effectively partitions* events into two sets, $E(X)$ and $E - E(X)$, and assigns a value $X(e)$ to events $e \in E(X)$.

Example 2. *Consensus specification: TwoThirds*

Let P and D be the classes of events with headers $propose$ and $decide$, respectively. Then the *safety specification* of a consensus protocol is the conjunction of two propositions on (extended) event-orderings, called *agreement* (all decision events have the same value) and *responsiveness* (the value decided on must be one of the values proposed):

$$\forall e_1, e_2 : E(D).\ D(e_1) = D(e_2)$$
$$\forall e : E(D).\ \exists e' : E(P).\ e' < e\ \wedge\ D(e) = P(e')$$

We can prove safety and the following *liveness property* about TwoThirds. We say that activity in the protocol *contracts to a subset* S of exactly $2f + 1$ processes if these processes all vote in election n say at $vt(n)_1, ..., vt(n)_k$ for $k = 2f + 1$ and collect these votes at $c(n)_1, ..., c(n)_k$, and all vote again in election $n + 1$ at $vt(n + 1)_1, ..., vt(n + 1)_k$, and collect at $c(n + 1)_1, ..., c(n + 1)_k$. In this case, these processes in S all decide in round $n + 1$ for the value given by f applied to the collected votes. This is a *liveness* property. If exactly f processes fail, then the activity of the group G contracts to some S and decides. Likewise if the message traffic is such that f processes are delayed for an election, then the protocol contracts to S and decides. This fact shows that the TwoThirds protocol is *non-blocking*, i.e. from any state of the protocol, there is a path to a decision. We can construct the path to a decision given a set of f processes that we delay. We also proved safety and liveness of a variant of this protocol that can converge to consensus faster. In this variant, if P_i receives a vote $\langle e, v\rangle$ from a higher election, $e > el_i$, then P_i joins that election by setting $el_i := e; v_i := v$; and then going to step 2.

A. Realizers and Strong Realizers

If ψ is a proposition about event orderings, e.g. liveness, we say that a system *realizes* ψ, if the event-ordering of any run of the system satisfies ψ. We extend the "proofs-as-programs" paradigm to "proofs-as-processes" for distributed computing by making constructive proofs that requirements are *realizable*. For compositional reasoning, it is desirable to create a *strong realizer* of requirement ψ—a system that realizes ψ *in any context*. Formally, system S is a strong realizer of ψ if the event-ordering of any run of a system S' such that $S \subseteq S'$, satisfies ψ. If S_1 is a strong realizer of ψ_1 and S_2 is a strong realizer of ψ_2, then $S_1 \cup S_2$ is a strong realizer of $\psi_1 \wedge \psi_2$. One of our main tools is that propagation rules like those used in the consensus example have strong realizers. A realizer for a propagation rule $A \overset{f}{\Rightarrow} B@g$ is a set of components that can each, as a (computable) function of the history of inputs at its location, recognize, and compute the value v of events in class A that occur there and send

messages that will eventually result in an events in class B with value $f(v)$ at each location in $g(v)$. We call the classes A that can be so recognized *programmable*. Basic event classes are programmable, and the set of programmable event classes is closed under a variety of *combinators*. Thus many classes can be automatically shown to be programmable, and their recognizers generated automatically, see [Bic09]. If B is a basic class and if we have *reliable message delivery*, then a component may cause an event in B by placing a message with an appropriate header on its external part. A rule, $A \Rightarrow B$ is *programmable-basic* (PB) if A is programmable and B is basic. *Under the assumption of reliable message delivery, every PB-rule is realizable.*

Reliable message delivery is an assumption about the environment. In this case, the assumption is a *fairness assumption* on the choices the environment makes, stating that all messages in the external part of a component will eventually be chosen. We weaken this assumption by allowing some components to suffer *send omission faults*. Under that assumption, parameterized by a set of locations, F, called the *fail-set*, every message on the external part of a component whose location is not in F, will eventually be delivered. If send omissions are allowed, not every PB-rule is realizable, but the restricted rule $A|(\neg F) \Rightarrow B$ is realizable, when $A \Rightarrow B$ is PB, and $A|(\neg F)$ is the class of A-events whose location is not in the fail-set. A fault-tolerant protocol like TwoThirds can be described by such restricted rules, and proved correct under appropriate assumptions on the size of the fail-set. A PB-rule $A \Rightarrow B$ is also strongly realizable. Some desirable properties of protocols like consensus do not follow from conjunctions of PB-rules alone. We also need some *propagation constraints*, of the form $A \overset{f}{\Leftarrow} B@g$. A realizer constructed for $A \overset{f}{\Rightarrow} B@g$ will generate B-events only from A-events, so it will also realize $A \overset{f}{\Leftarrow} B@g$.

III. Attack Tolerance

We assume that our systems will be attacked. We will protect protocols by formally generating a large number of *logically equivalent variants*, stored in an *attack response library*. Each variant uses distinctly different code which a system under attack can install *on-the-fly* to replace compromised components. Each variant is known to be equivalent and correct. We express threatening features of the environment formally and discriminate among the different types. We can do this in our new GPM model because *the environment is an explicit component about which we can reason.* This capability allows us to study in depth how diversity works to defend systems. We can implement attack scenarios as part of our experimental platform. It is interesting that we can implement a *constructive version* [RC08] of the famous Fischer/Lynch/Paterson result [FLP85] that shows how an attacker can keep any deterministic fault-tolerant consensus protocol from achieving consensus.

A. Synthetic Code Diversity

We introduce diversity at all levels of the formal code development (synthesis) process starting at a very high level of abstraction. For example, in the TwoThirds protocol, we can use different functions f, alter the means of collecting Msg_i, synthesize variants of the protocol, alter the data types, etc. We are able to create multiple provably correct versions of protocols at each level of development, e.g. compiling TwoThirds into Java, Erlang, and $F^{\#}$. The higher the starting point for introducing diversity, the more options we can create. We can also inject code diversity into the fully automatic verification of authentication protocols in *Protocol Composition Logic* (PCL) [DDMR07] implemented in our system. These synthetic diversity techniques generate a large set of components, each of which is associated with a "genotype" that describes the parameters (such as choice of algorithm, choice of data structure, choice of implementation language) used to generate components of its "species".

B. Formalizing the Environment

We can precisely describe how diversification and reconfiguration respond to threatening features of the environment because we can *express some threats formally* and discriminate among the different kinds as is done in the *Nysiad* system [HvRBD08] to defend against Byzantine attacks. This capability will allow us to study how diversity defends systems. For example, we have studied *message order attacks* in which the environment delays key messages in consensus protocols to keep the system from deciding in a timely manner. This is an instance of the general phenomenon uncovered by Fischer, Lynch, and Paterson [FLP85] that consensus algorithms can be systematically defeated, as we discuss next.

a) The Environment as Adversary: The standard version of the Fischer/Lynch/Paterson theorem is that no deterministic algorithm can solve the consensus problem for a group of process in which at least one process might fail. This is a negative statement, producing only a contradiction, yet implicit in all proofs is an *imagined construction* of a nonterminating execution in which no process decides, they "waffle" endlessly. It appears that no such explicit construction could be carried out following the method of the classical proof because there isn't enough information given with the protocol.

The key to being able to build the nonterminating execution is to provide more information, which was done in [RC08] by introducing the notion of *effective nonblocking*. Effective nonblocking is a natural concept when protocols are verified using constructive logic. **P** is called *effectively nonblocking* if from any reachable global state s of an execution of **P** and any subset Q of $n - f$ non-failed processes, *we can find* an execution α from s using Q and a process P_α in Q which decides a value v. Constructively this means that we have a *computable function*, $wt(s, Q)$, which produces an execution α and a state s_α in which a process, say P_α decides a value v. **Theorem (CFLP):** Given any deterministic

effectively nonblocking consensus procedure **P** with more than two processes and tolerating a single failure, we can effectively construct a nonterminating execution of it. Let the function produced by this proof be **flpc**, then for a consensus procedure such as the *TwoThirds* protocol given above and its nonblocking proof nb, we have that the environment can use **flpc**(nb) to create a message-order attack that will prevent TwoThirds from deciding.

IV. Conclusion

We are exploring how to build distributed systems that are attack-tolerant by design. The key idea is to implement systems that can respond to attacks by modifying their code in a provably safe way. We believe that the more code variants we can produce, the more resistant systems are to attack. We have found ways to automatically produce many provably equivalent variants of components using formal synthesis. Variation arises from different choices made during synthesis. We start at a very high level of abstraction by formally proving that specifications are achievable. By starting at such a high level, we discovered more correct options than possible by less technically advanced methods. This discovery reveals new reasons for working formally at high levels of abstraction.

References

[ABC06] Stuart Allen, Mark Bickford, Robert Constable, Richard Eaton, Christoph Kreitz, Lori Lorigo, and Evan Moran. Innovations in Computational Type Theory using Nuprl, *Journal of Applied Logic*, Elsevier Science, 428-469, 2006.

[Abr99] Uri Abraham. *Models for Concurrency*, Gordon and Breach Science Publishers, 1999.

[AW04] Hagit Attiya and Jennifer Welch. *Distributed Computing*, 2nd Edition, Wiley, 2004.

[Bic09] Mark Bickford. Component Specification Using Event Classes. *Lecture Notes in Computer Science*, Vol 5582, 140-155, 2009.

[BC10a] Mark Bickford and Robert Constable Generating Event Logics with Higher-Order Processes as Realizers, Computer Science Technical Report, Cornell University, 2010.

[CB08] Robert Constable and Mark Bickford. Formal Foundations of Computer Security, *NATO Science for Peace and Security Series D: Information and Communication Security*, Vol 14, 29-52, 2008.

[CPS93] Rance Cleaveland, Joachim Parrow, and Bernhard Steffen. The Concurrency Workbench: A Semantics Based Tool for the Verification of Concurrent Systems, *ACM Transactions on Programming Languages and Systems*, 15, 36-72, 1993.

[RC08] Robert Constable. Effectively Nonblocking Concensus Procedures Can Execute Forever - a Constructive Version of FLP, Cornell University Tech Report Ref Number 11512, 2008.

[DDMR07] A. Datta, A. Derek, J.C. Mitchell, A. Roy. Protocol Composition Logic, *Electronic Notes in Theoretical Computer Science*, 172, 311-358, 2007.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faculty process. *JACM*, Vol 32, 374–382, 1985.

[HvRBD08] C. Ho, R. van Renesse, M. Bickford, D. Dolev. Nysiad: practical protocol transformation to tolerate Byzantine failures, *Proceedings of the 5th USENIX Symposium on Networked Sys Design and Impl*,175-188, 2008.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system, *Comm ACM*, 21(7), 558-65, 1978.

[Lam01] Leslie Lamport. Paxos made simple, *ACM SIGACT News*, 4, December, 2001, 18-25.

[LKvR+99] X. Liu, C. Kreitz, R. van Renesse, J.Hickey, M. Hayden, K. Birman, R. Constable. Building reliable, high-performance communication systems from components, *ACM Symposium on Operating Systems Principles (SOSP)*, ACM Press, 80-92, 1999.

[Lyn96] Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, San Mateo, CA, 1996.

[Mil09] Robin Milner. *The Space and Motion of Communicating Agents*, Cambridge University Press, 2009.

[Win89] Glynn Winskel. An introduction to event structures, *LNCS* 345, Springer, NY, 364-397, 1989.