Event Structures over Process Models

Robert Constable
Cornell University

Abstract

Building Event Structures over Higher-Order Processes as Realizers

Robert Constable
Cornell University

Using a constructive Logic of Events based on 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 We say that the programs realize the specifications. We have used this proofs-as-processes method to build fault-tolerant protocols, adaptive protocols, and provably secure protocols. Recently we have created versions of Paxos this way.

This system development capability is based on a constructive semantics for assertions in our “standard” Logic of Events using the concept of event structures from Winskel adapted to executions of process in the standard model of asynchronous message passing computation.

Abstract

The realizers are state machines similar to IOA which can easily be compiled into appropriate programming languages such as Java, Erlang, F#, etc. Critical to the practical success of this methodology is the use of event classes to specify computing tasks at a high level of abstraction that can be refined automatically to processes.

In 2010 we substantially extend this synthesis/verification/development method so that it applies to a very general notion of process of the kind used in process algebras (e.g. the higher-order pi-calculus, Petri nets, CCS, CSP, etc.) as well for the standard process model used in the Logic of Events mentioned above. Our generalization enables the synthesis of correct-by-construction processes over a wide variety of process models by extracting them as distributed realizers for specifications event logic.

The lecture will illustrate this development method on a consensus protocol, explain the basic concepts and point to some interesting questions.

The Story

My Cornell group and I are known for showing how to treat constructive proofs as programs and treat formal mathematics as a programming language. This has become a flourishing enterprise in certain places, e.g. Harvard, Penn, CMU, Edinburgh, etc.

Since the mid 90's we have wanted to extend this method to proofs as processes, building protocols from constructive proofs that specifications are realizable in a distributed formal theory.

The Story continued

We started by using IOA as our internal model of processes and a distributed database under our proof assistant. In 2003 we modified IOA to Message Automata and built an event logic around this model. These MA used frame conditions to render composition as union.

Year by year as we tackled harder protocols, we have been forced to be more and more abstract in order to complete the proofs and extract protocols, and we are being forced to replicate the database.

The Story continued

Now we can create a variety of protocols from proofs, e.g. consensus (e.g. Paxos, 2/3), authentication, group membership, etc.

We found advantages of starting very abstractly, e.g. we can generate many provably correct variants at the same time, providing attack-tolerance.

The Story continued

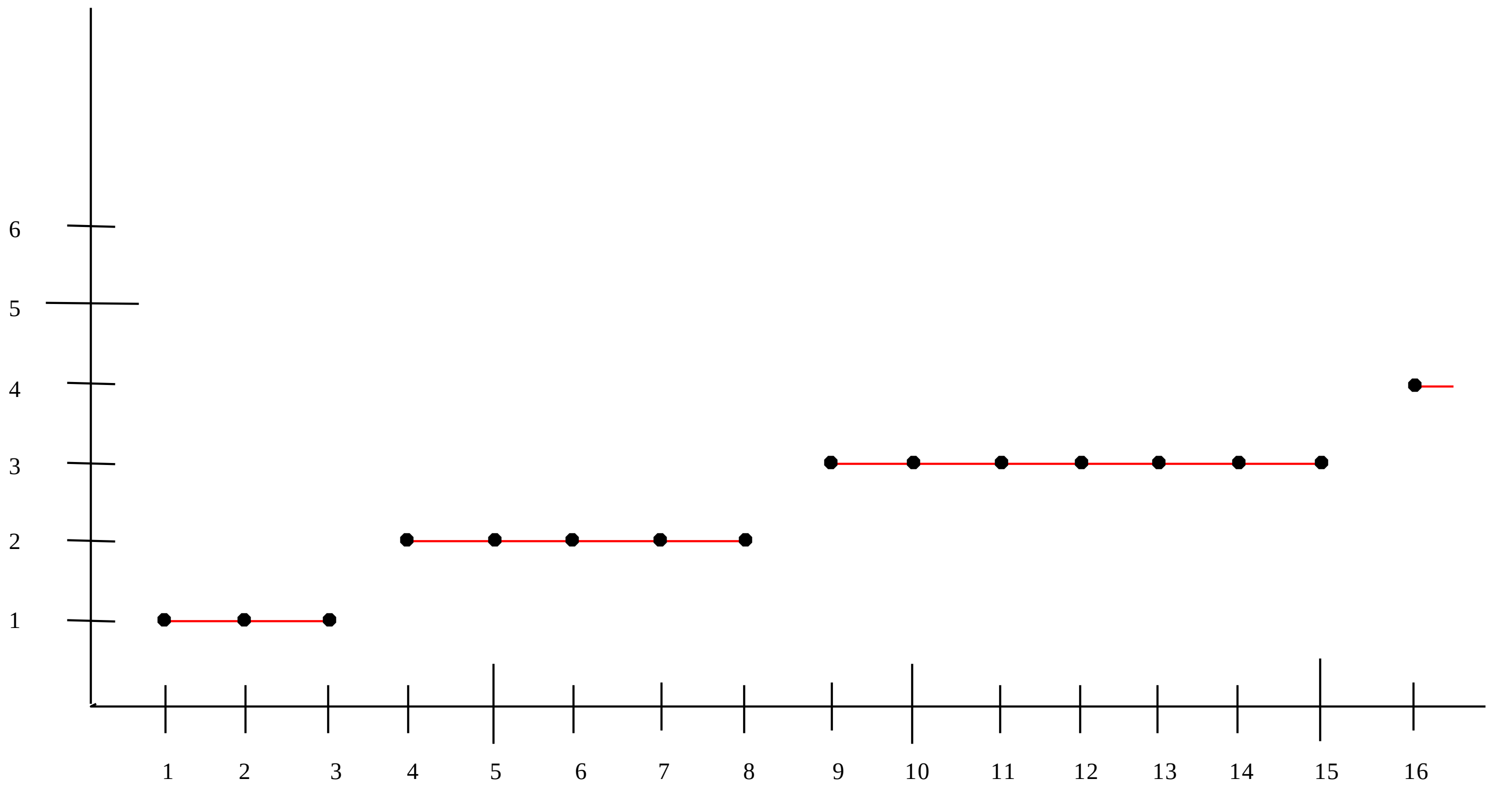
Our constructive proofs of consensus require proofs of non-blocking. I discovered that FLP can be proved constructively for effectively non-blocking protocols.

From Constructive FLP we can build an unbeatable adversary (attacker) against deterministic consensus.

Outline

1. Example of proofs-as-programs
2. Example of proofs-as-processes
3. Event class specification – most abstract
4. Attack-tolerance and FLP adversary

Integer Square Root



Proof of Root Theorem

$\forall \mathtt{n} : \mathbb{N}.\ \exists \mathtt{r} : \mathbb{N}.\ \mathtt{r}^2 \leq \mathtt{n} < (\mathtt{r} + 1)^2$

BY `allR`

$\mathtt{n} : \mathbb{N}$

$\vdash \exists \mathtt{r} : \mathbb{N}.\ \mathtt{r}^2 \leq \mathtt{n} < (\mathtt{r} + 1)^2$

BY <u>NatInd</u> 1

. . . . .induction case.....

$\vdash \exists \mathtt{r} : \mathbb{N}.\ \mathtt{r}^2 \leq 0 < (\mathtt{r} + 1)^2$

BY `existsR` $\lceil 0 \rceil$ THEN `Auto`

. . . . .induction case.....

$\boldsymbol{i} : \mathbb{N}^+,\ \mathtt{r} : \mathbb{N},\ \mathtt{r}^2 \leq \boldsymbol{i} - 1 < (\mathtt{r} + 1)^2$

$\vdash \exists \mathtt{r} : \mathbb{N}.\ \mathtt{r}^2 \leq \boldsymbol{i} < (\mathtt{r} + 1)^2$

BY `Decide` $\left\lceil (\underline{\mathtt{r}} + 1)^2 \leq \boldsymbol{i} \right\rceil$ THEN `Auto`

Proof of Root Theorem (cont.)

. . . . .Case 1.....

$i : \mathbb{N}^{+},\ \mathtt{r} : \mathbb{N},\ \mathtt{r}^2 \le i - 1 < (\mathtt{r} + 1)^2,\ (\mathtt{r} + 1)^2 \le i$

$\vdash \exists \mathtt{r} : \mathbb{N}.\ \mathtt{r}^2 \le i < (\mathtt{r} + 1)^2$

BY `existsR` $\lceil \underline{\mathtt{r}} + 1 \rceil$ THEN `Auto'`

. . . . .Case 2.....

$i : \mathbb{N}^{+},\ \mathtt{r} : \mathbb{N},\ \mathtt{r}^2 \le i - 1 < (\mathtt{r} + 1)^2,\ \neg\left((\mathtt{r} + 1)^2 \le i\right)$

$\vdash \exists \mathtt{r} : \mathbb{N}.\ \mathtt{r}^2 \le i < (\mathtt{r} + 1)^2$

BY `existsR` $\lceil \underline{\mathtt{r}} \rceil$ THEN `Auto`

The Root Program Extract

Here is the extract term for this proof in ML notation with proof terms (pf) included:

$$
\begin{array}{l}
\text{let } \mathtt{rec}\ \mathtt{sqrt}\ \boldsymbol{i} = \\
\quad \text{if } \boldsymbol{i} = 0 \text{ then } \langle 0, \mathtt{pf}_0 \rangle \\
\qquad \text{else let } \langle \mathtt{r}, \mathtt{pf}_{i-1} \rangle = \mathtt{sqrt}(\boldsymbol{i} - 1) \\
\qquad \text{in if } (\mathtt{r} + 1)^2 \leq \mathtt{n} \text{ then } \langle \mathtt{r} + 1, \mathtt{pf}_i \rangle \\
\qquad\quad \text{else } \langle \mathtt{r}, \mathtt{pf}_i\texttt{'} \rangle
\end{array}
$$

A Recursive Program for Integer Roots

Here is a very clean functional program

$$
\begin{aligned}
r(n) := \textbf{if } & n = 0 \textbf{ then } 0 \\
& \textbf{else } \text{let } r_0 = r\,(n\text{-}1) \textbf{ in} \\
& \textbf{if } (r_0 + 1)^2 \leq n \textbf{ then } r_0 + 1 \\
& \qquad\qquad\qquad\quad \textbf{else } r_0 \textbf{ fi} \\
& \textbf{fi}
\end{aligned}
$$

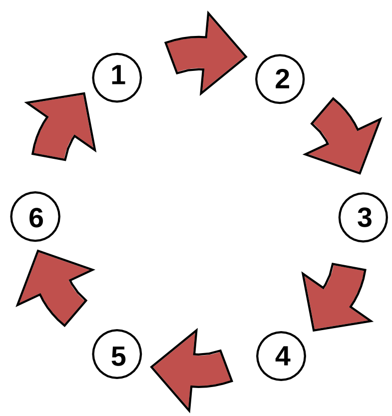
This program is close to a declarative mathematical description of roots given by the following theorem.

Outline

2. Example of proofs-as-processes
3. Event class specification – most abstract
4. Attack-tolerance and FLP adversary

Specification for Leader Election in a Ring

Given a Ring R of Processes with Unique Identifiers (uid's)



Let $n(\mathtt{i}) = dst(out(\mathtt{i}))$, the next location

Let $p(\mathtt{i}) = n^{-1}(\mathtt{i})$, the predecessor location

Let $d(\mathtt{i},j) = \mu k \geq 1.\ n^{k}(\mathtt{i}) = j$, the distance from $\mathtt{i}$ to j

Note $\mathtt{i} \neq p(j) \Rightarrow d(\mathtt{i},p(j)) = d(\mathtt{i},j)-1$.

Specification, continued

$$\text{Leader}(R,es) == \exists\, ldr{:}\, R.\ \ (\exists e@ldr.\ kind(e)=leader)\ \&$$
$$(\forall i{:}R.\ \forall e@i.\ kind(e)=leader\ \Rightarrow\ i=ldr)$$

Theorem $\forall\ R{:}List(Loc).\ Ring(R)$

$\exists\ D{:}Dsys(R).\ Feasible(D)\ \&$

$\forall es{:}\ ES.\ Consistent(D,es).\ Leader(R,es)$

Decomposing the Leader Election Task

Let LE(R,es) == $\forall$i:R.

1. $\exists$e. kind(e)=rcv(out(i), <vote,uid(i)>)

2. $\forall$e'. kind(e)=rcv (in(i), <vote,u>) $\Rightarrow$

   $\left(u>uid(i) \;\Rightarrow\; \exists e'.kind(e')=rcv\ (out(i),<vote,u>)\right)$

3. $\forall e'.\ [\ \ (kind(e')=rcv(out(i), <vote,uid(i)>)) \vee$

   $\exists e.\ \left(kind(e)=rcv(in(i), <vote,u>) \& \left(e < e' \ \&\ u > uid(i)\right)\right)\ \ ]$

4. $\forall$e@i. kind(e)=rcv(in(i),uid(i)). $\exists$e′@i. kind(e′)=leader

5. $\forall$e@i. kind(e)=leader. $\exists$e@i. kind(e)=rcv $\left(in(i), <vote,uid(i)>\right)$

Realizing Leader Election

Theorem $\forall \mathtt{R{:}List(Loc)}\,.\,\mathtt{Ring(R)}$

$\exists \mathtt{D{:}Dsys(R)}\,.\,\mathtt{Feasible(D)}\,.$

$\forall \mathtt{es{:}Consistent(D,\ es)}\,.\ \big(\mathtt{LE(R,\ es)} \Rightarrow \mathtt{Leader(R,\ es)}\big)$

Proof: Let $\mathtt{m} = \mathtt{max}\,\{\mathtt{uid(i)} \mid \mathtt{i} \in \mathtt{R}\}$, then $\mathtt{ldr} = \mathtt{uid}^{-1}(\mathtt{m})$.

We prove that $\mathtt{ldr} = \mathtt{uid}^{-1}(\mathtt{m})$ using three simple lemmas.

Intuitive argument that a leader is elected

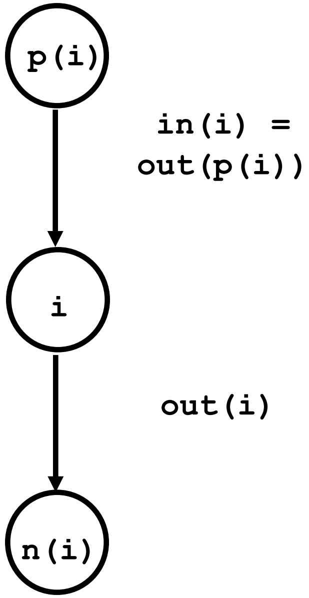
1. Every i will get a vote from predecessor for the predecessor.

2. When a process i gets a vote u from its predecessor with $u > \text{uid}(i)$ it sends it on.

3. Every rcv is either vote of predecessor $\text{rcv}_{\text{in(i)}}$ for itself or a vote larger than process id before.

4. If a process gets a vote for itself, it declares itself ldr.

5. If a processor declares ldr it got a vote for itself.



Lemmas

Lemma 1. $\forall \mathtt{i} : \mathtt{R}.\ \exists \mathtt{e} \mathbin{@} \mathtt{i}.\ \mathtt{kind}(\mathtt{e}) = \mathtt{rcv}\left(\mathtt{in}(\mathtt{i}), \langle \mathtt{vote}, \mathtt{ldr} \rangle\right)$

By induction on distance of `i` to `ldr`.

Lemma 2. $\forall \mathtt{i}, \mathtt{j} : \mathtt{R}.\ \forall \mathtt{e} \mathbin{@} \mathtt{i}.\ \mathtt{kind}(\mathtt{e}) = \mathtt{rcv}\left(\mathtt{in}(\mathtt{i}), \langle \mathtt{vote}, \mathtt{j} \rangle\right).$

$$\left(\mathtt{j} = \mathtt{ldr} \lor \mathtt{d}(\mathtt{ldr}, \mathtt{j}) < \mathtt{d}(\mathtt{ldr}, \mathtt{i})\right)$$

By induction on causal order of `rcv` events.

Lemma 3. $\forall \mathtt{i} : \mathtt{R}.\ \forall \mathtt{e}' \mathbin{@} \mathtt{i}.\ \left(\mathtt{kind}(\mathtt{e}') = \mathtt{leader} \Rightarrow \mathtt{i} = \mathtt{ldr}\right)$

If $\mathtt{kind}(\mathtt{e}') = \mathtt{leader}$, then by property 5, $\exists \mathtt{v} \mathbin{@} \mathtt{i}.\mathtt{rcv}\left(\mathtt{in}(\mathtt{i}), \langle \mathtt{vote}, \mathtt{uid}(\mathtt{i}) \rangle\right).$

Hence, by Lemma 2 $\mathtt{i} = \mathtt{ldr} \lor \left(\mathtt{d}(\mathtt{ldr}, \mathtt{i}) < \mathtt{d}(\mathtt{ldr}, \mathtt{i})\right)$

but the right disjunct is impossible.

Finally, from property 4, it is enough to know

$\exists \mathtt{e}.\mathtt{kind}(\mathtt{e}) = \mathtt{rcv}\left(\mathtt{in}(\mathtt{ldr}), \langle \mathtt{vote}, \mathtt{uid}(\mathtt{ldr}) \rangle\right)$

which follows from Lemma 1.

QED

Realizing the clauses of LE(R,es)

We need to show that each clause of `LE(R,es)` can be implemented by a piece of a distributed system, and then show the pieces are compatible and feasible.

We can accomplish this very logically using these Lemmas:

- Constant Lemma
- Send Once Lemma
- Recognizer Lemma
- Trigger Lemma

Leader Election Message Automaton

state $me : \square$; initially $uid(i)$

state $done : B$; initially $false$

state $x : B$; initially $false$

action $vote$; precondition $\neg done$

  effect $done := true$

  sends $[msg\left(out(i), vote, me\right)]$

action $rcv_{in(i)}(vote)(v) : \square$;

  sends if $v > me$ then $[msg\left(out(i), vote, v\right)]$ else[]

  effect $x :=$ if $me = v$ then $true$ else x

action $leader$; precondition $x = true$

only $rcv_{in(i)}(vote)$ affects x

only $vote$ affects $done$

only $\{vote, rcv_{in(i)}(vote)\}$ sends $out\ (i), vote$

Outline

1. Example of proofs-as-programs
2. Example of proofs-as-processes
3. Event class specification – more abstract
4. Attack-tolerance and FLP adversary

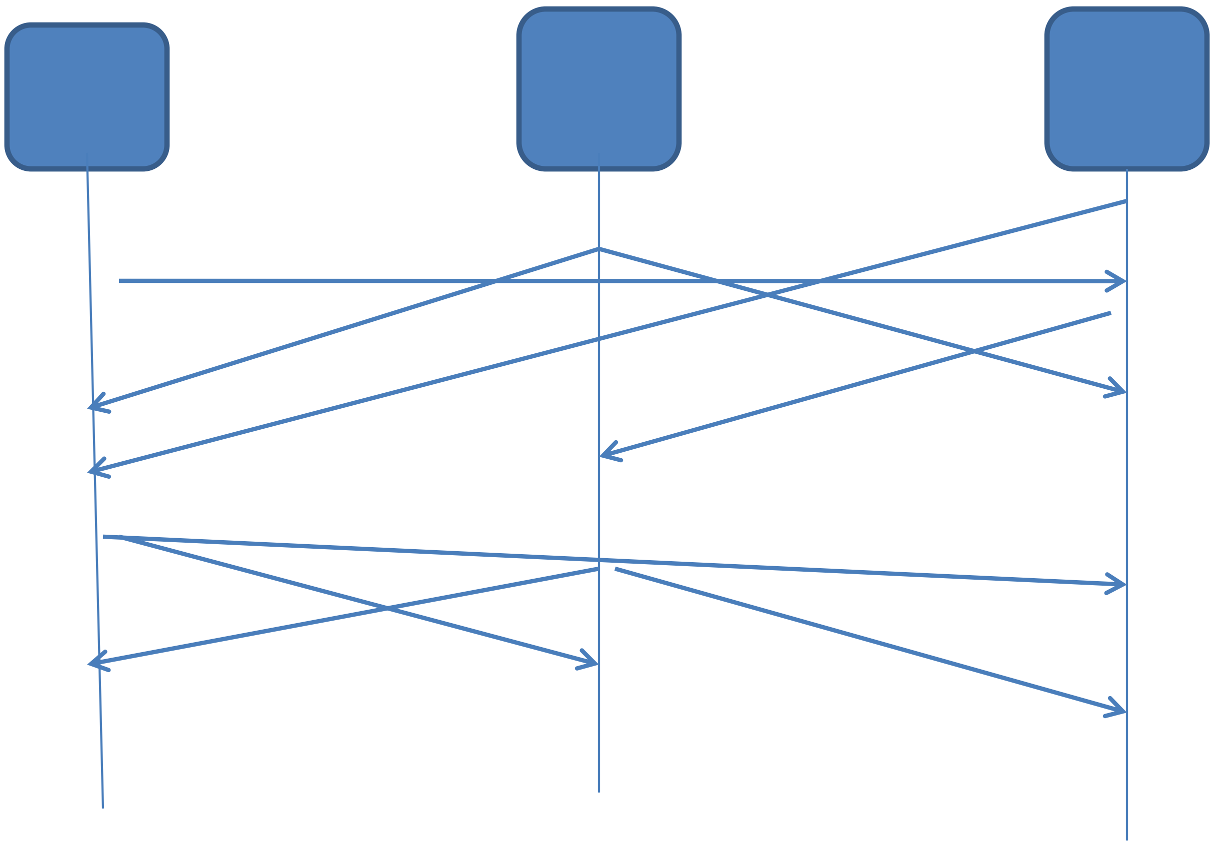
Consensus is a Good Example

In modern distributed systems, e.g. the Google file system, clouds, etc., reliability against faults (crashes, attacks) is achieved by replication.

Consensus is used to coordinate write actions to keep the replicas identical. It is a critical protocol in modern systems used by IBM, Google, Microsoft, Amazon, EMC, etc.

Requirements of Consensus Task

Use asynchronous message passing to decide on a value.



Logical Properties of Consensus

P1: If all inputs are unanimous with value v, then any decision must have value v.

All v:T. (If All e:E(Input). Input(e) = v then
All e:E(Decide). Decide(e) = v)

Input and Decide are event classes that effectively partition the events and assign values to them. The events are points in abstract space/time at which "information flows." More about this just below.

Logical Properties continued

P2: All decided values are input values.

All e:E(Decide). Exists e':E(Input).

e' < e & Decide(e) = Input(e')

We can see that P2 will imply P1, so we take P2 as part of the requirements.

Event Classes

If X is an event class, then E(X) are the events in that class. Note E(X) effectively partitions all events E into E(X) and E-E(X), its complement.

Every event in E(X) has a value of some type T which is denoted X(e). In the case of E(Input) the value is the typed input, and for E(Decide) the value is the one decided.

Events

Formally the type E of events is defined relative to the computation model which includes a definition of processes.

The events are the points of space/time at which information is exchanged. The information at an event e is info(e).

Further Requirements for Consensus

The key safety property of consensus is that all decisions agree.

P3: Any two decisions have the same value.

This is called agreement.

All e1,e2: E(Decide). Decide(e1) = Decide(e2).

Specific Approaches to Consensus

Many consensus protocols proceed in rounds, voting on values, trying to reach agreement. We have synthesized two families of consensus protocols, the 2/3 Protocol and the Paxos Protocol families.

We structure specifications around events during the voting process, defining E(Vote) whose values are pairs <n,v>, a ballot number, n, and a value, v.

Properties of Voting

Suppose a group G of n processes, Pi, decide by voting. If each Pi collects all n votes into a list L, and applies some deterministic function f(L), such as majority value or maximum value, etc., then consensus is trivial in one step, and the value is known at each process in the first round – possibly at very different times.

The problem is much harder because of possible failures.

Fault Tolerance

Replication is used to ensure system availability in the presence of faults. Suppose that we assume that up to f processes in a group G of n might fail, then how do the processes reach consensus?

The TwoThirds method of consensus is to take n = 3f +1 and collect only 2f+1 votes on each round, assuming that f processes might have failed.

Example for f = 1, n = 4

Here is a sample of voting in the case T = {0,1}.

0	0	1	1	inputs

0 _11	_011	001_	00_1	collected votes
1	1	0	0	next vote

00_1	001_	0_11	_011
0	0	1	1

where f is majority voting, first vote is input

Specifying the 2/3 Method

We can specify the fault tolerant 2/3 method by introducing further event classes.

E(Vote), E(Collect), E(Decide)

E(Vote): the initial vote is the <0,input value>, subsequent votes are <n,f(L)>

E(Collect): collect 2f+1 values from G into list L

E(Decide): decide v if all collected values are v

The Hard Bits

The small example shows what can go wrong with 2/3. It can waffle forever between 0 and 1, thus never decide.

Clearly if there is are decide events, the values agree and that unique value is an input.

Can we say anything about eventually deciding, e.g. liveness?

Liveness

If f processes eventually fail, then our design will work because if f have all failed by round r, then at round r+1, all alive processes will see the same 2f+1 values in the list L, and thus they will all vote for v' = f(L), so in round r+2 the values will be unanimous which will trigger a decide event.

Example for f = 1, n = 4

Here is a sample of voting in the case T = {0,1}.

0	0	1	1	inputs
0 01_	001_	001_	_011	collected votes
0	0	0	1	next vote

000_	00_1	0_01	_001
0	0	0	0

where f is majority voting, first vote is input, round numbers omitted.

Safety Example

We can see in the f = 1 example that once a process Pi receives 2/3 unanimous values, say 0, it is not possible for another process to over turn the majority decision.

Indeed this is a general property of a 2/3 majority, the remaining 1/3 cannot overturn it even if they band together on every vote.

Safety Continued

In the general case when voting is not by majority but using f(L) and the type of values is discrete, we know that if any process Pi sees unanimous value v in L, then any other process Pj seeing a unanimous value v' will see the same value, i.e. v = v' because the two lists, Li and Lj at round r must share a value, that is they intersect.

Synthesizing the 2/3 Protocol from a Proof of Design

We can formally prove the safety and liveness conditions from the event logic specification given earlier.

From this formal proof of design, **pf**, we can automatically extract a protocol, first as an abstract process, then by verified compilation, a program in Java or Erlang.

The Synthesized 2/3 Protocol

Begin r:Nat, decided_i, vote_i: Bool,
r = 0, decided_i = false, vi = input to Pi; vote_i = vi

Until decided_i **do**:

1. r := r+1
2. **Broadcast** vote <r,vote_i> to group G
3. **Collect** 2f+1 round r votes in list L
4. vote_i := majority(L)
5. **If** unanimous(L) **then** decided_i := true

End

Abstract Process Model

$M(P) == (Atom\ List) \times (T + P)$

$E(P) == (Loc \times M(P))\ List$

$F(P) = M(P) \rightarrow (P \times E(P))$

It is easy to show that *M and E are continuous type functions* and that *F is weakly continuous. Thus for*

$Process == corec(P.\ F(P))$

$Msg == M(Process)$ *and* $Ext == E(Process)$

we conclude *Process is a subtype of F(Process),*

$Process \subseteq Msg \rightarrow Process \times Ext$

Executing Systems of Processes

The environment chooses which messages will be delivered. A run of a system is an unbounded sequence of pairs <sys,choice>.

From a run of a system, we can build event structures with locations and causal order.

Event Orderings over Runs

An event ordering of a run R is a collection of events E, a function loc giving the location of the event, a well founded causal order < on events, and info, the information conveyed by an event: <E, loc, <, info>

The events are pairs <x,n> at which location x receives a message at step n of the run.

Event Structures over Runs

Event structures include the operations

x **when** e and x **after** e

for state variable x an events e, and the axiom

not first(e) implies (x when e = x after pred(e))

Diversity

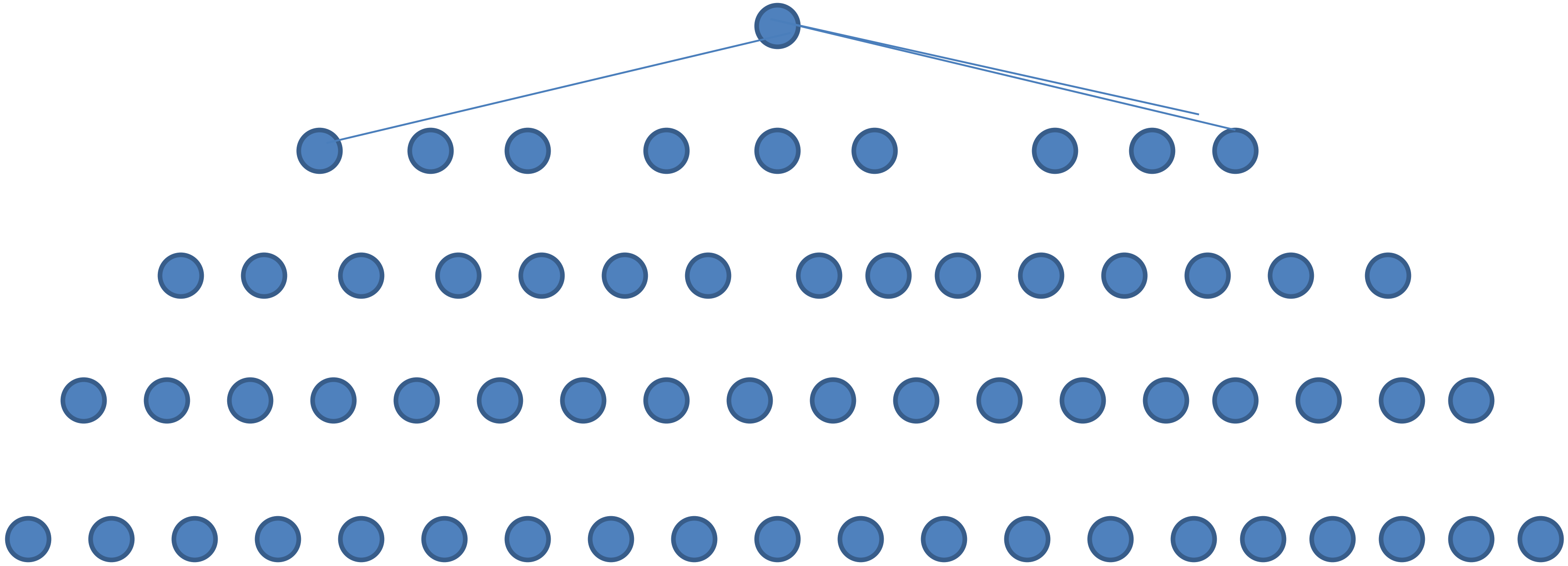
When we prove properties of a design, there are many options at several steps, and we are able to create multiple proofs at low additional cost. In the process we create new designs.

For example, for the 2/3 protocol, Mark Bickford found a variant that is faster by varying the design proof, as mentioned in our paper – he varies the collection method.

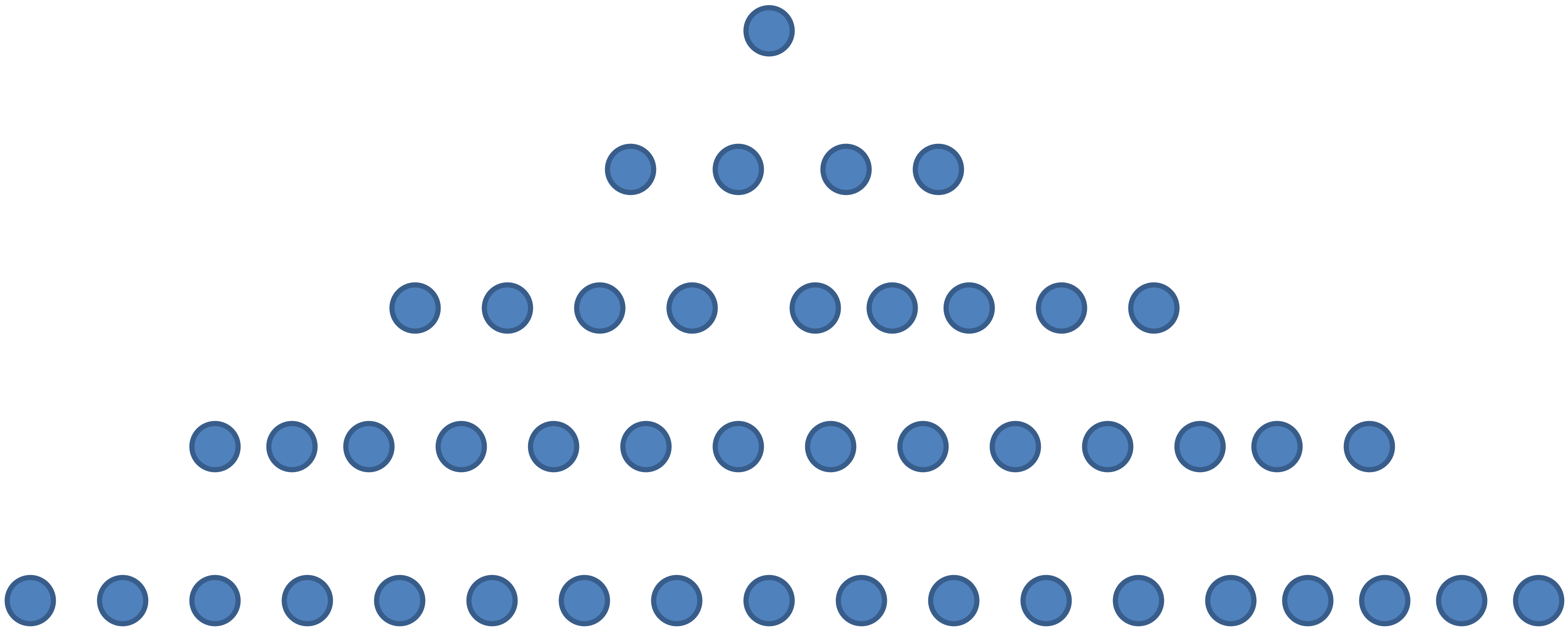
Diversity at the Level of Proof

Multiple formal proofs are “simultaneously” generated. We illustrate this by viewing a proof as a tree generated top down.

Illustrating Multiple Proofs



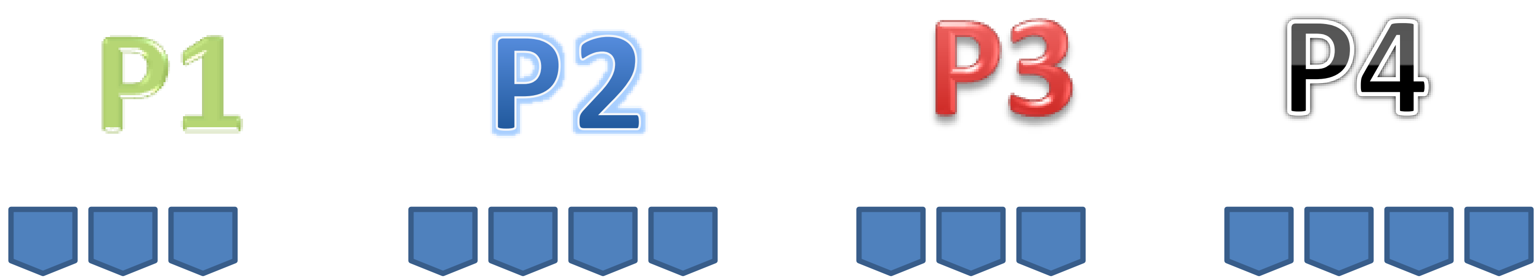
Illustrating Multiple Proofs



P1 P2 P3 P4

Data Structure Diversity

Assuming there are four abstract protocols derived from the proof trees. For each of them it is possible to implement with different data structures, e.g. list, array, tree, set, etc.

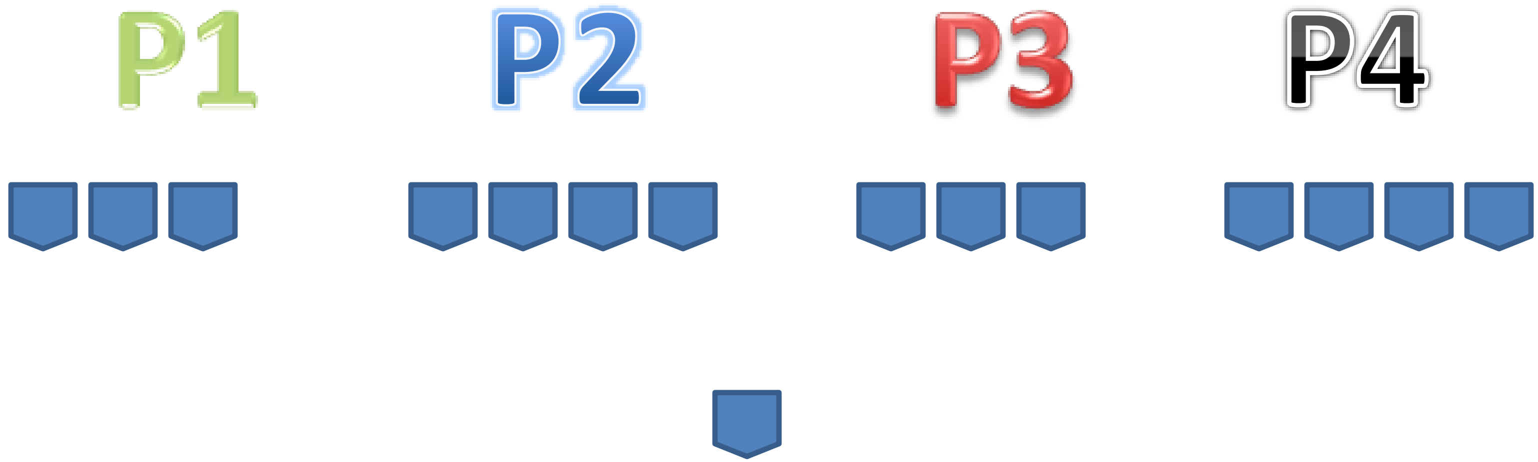


Programming Language Diversity

We can translate abstract programs into common programming languages such as Java, Erlang, C++, or F#. So far we use only Java and Erlang.

Combining all levels of diversity we are able to generate over 200 variants of a protocol in the best case.

Language Diversity



Java ML Erlang F#

4 protocols, 14 options in 4 languages, offers over 200 variants

Outline

1. Example of proofs-as-programs
2. Example of proofs-as-processes
3. Event class specification – most abstract
4. Attack-tolerance and FLP adversary

Role of the Environment

All distributed computing models must have a component that determines when messages between processes are delivered. We call this the environment. It introduces uncertainty into the model and determines the schedule of events.

A Fundamental Theorem of about the Environment

The Fischer/Lynch/Paterson theorem (FLP85) about the computing environment says:

it is not possible to deterministically guarantee consensus among two or more processes when one of them might fail.

We have seen the possibility of this with the 2/3 Protocol which could waffle between choosing 0 or 1. The environment can act as an adversary to consensus by managing message delivery.

The Environment as Adversary

In the setting of synthesizing protocols, I have shown that the FLP result can be made constructive (CFLP). This means that there is an algorithm, env, which given a potential consensus protocol P and a proof pf that it is nonblocking can create message ordering and a computation based on it, env(P,pf), in which P runs forever, failing to achieve consensus.

Perfect Attacker

The algorithm env(P,pf) is the perfect "denial of service attacker" against any consensus protocol P that is sensible (won't block).

Note, 2/3 will block if it waits for n replies or if it refuses to change votes as rounds progress.

Defending Against the Perfect Attack

One way to defend against evn(P,pf) is to switch to another protocol P' if there appears to be an attack against P.

Definitions

P is called effectively nonblocking if from any reachable global state s of an execution of P and any subset Q of n – t nonfailed processes, we can find an execution from s using Q and a process P in Q which decides a value v.

Constructively this means that we have a computable function, wt(s,Q) which produces an execution and a state s in which a process, say P decides a value v.

Constructive FLP

Theorem (Constructive FLP): Given any deterministic effectively nonblocking consensus procedure P with two or more processes tolerating a single failure, we can effectively construct a non-terminating execution of it.

Key Lemma

One Step Lemma: Given any bivalent global state b of an effectively nonblocking consensus procedure P, and any process Pi, we can find a extension b’ of b which is bivalent via Qi.

FLP as a Corollary

The proof is to use the Initialization Lemma to find a bivalent starting state b and then use the One Step Lemma to create an unbounded sequence of bivalent states.

Corollary (FLP): There is no single-failure responsive, deterministic consensus algorithm (terminating consensus procedure) on two or more processes.

Corollary (Strong FLP)*: Given any nonblocking deterministic consensus procedure on two or more processes, it has a non-terminating execution.

Conclusion

We are exploring how to build distributed systems that are attack-tolerant by design.

The key idea is to implement systems that 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 will be to attack.

Conclusion

We initiate code generation at a very high level of abstraction by formally proving that designs are realizable.

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.

THE END

Definition

Attack-tolerant distributed systems change their protocols on-the-fly in response to apparent attacks from the environment; they substitute functionally equivalent components possibly more resistant to detected threats.

Definition

A system is built from components which consist of processes (protocols, algorithms).

system

component

process

Definition

A system is correct-by-construction if we create a correctness proof for it while creating the code. This happens if we synthesize the program from a constructive proof that the specification is realizable.

Main Result

We have found ways to automatically produce many provably equivalent variants of components using formal synthesis.

Variation arises from different choices made during the proof and code synthesis process starting from formal specifications.

A Discovery

In the course of this work we also discovered that it is possible to create undefeatable attackers for deterministic fault-tolerant consensus protocols.

Code diversity can protect against these attackers as well.

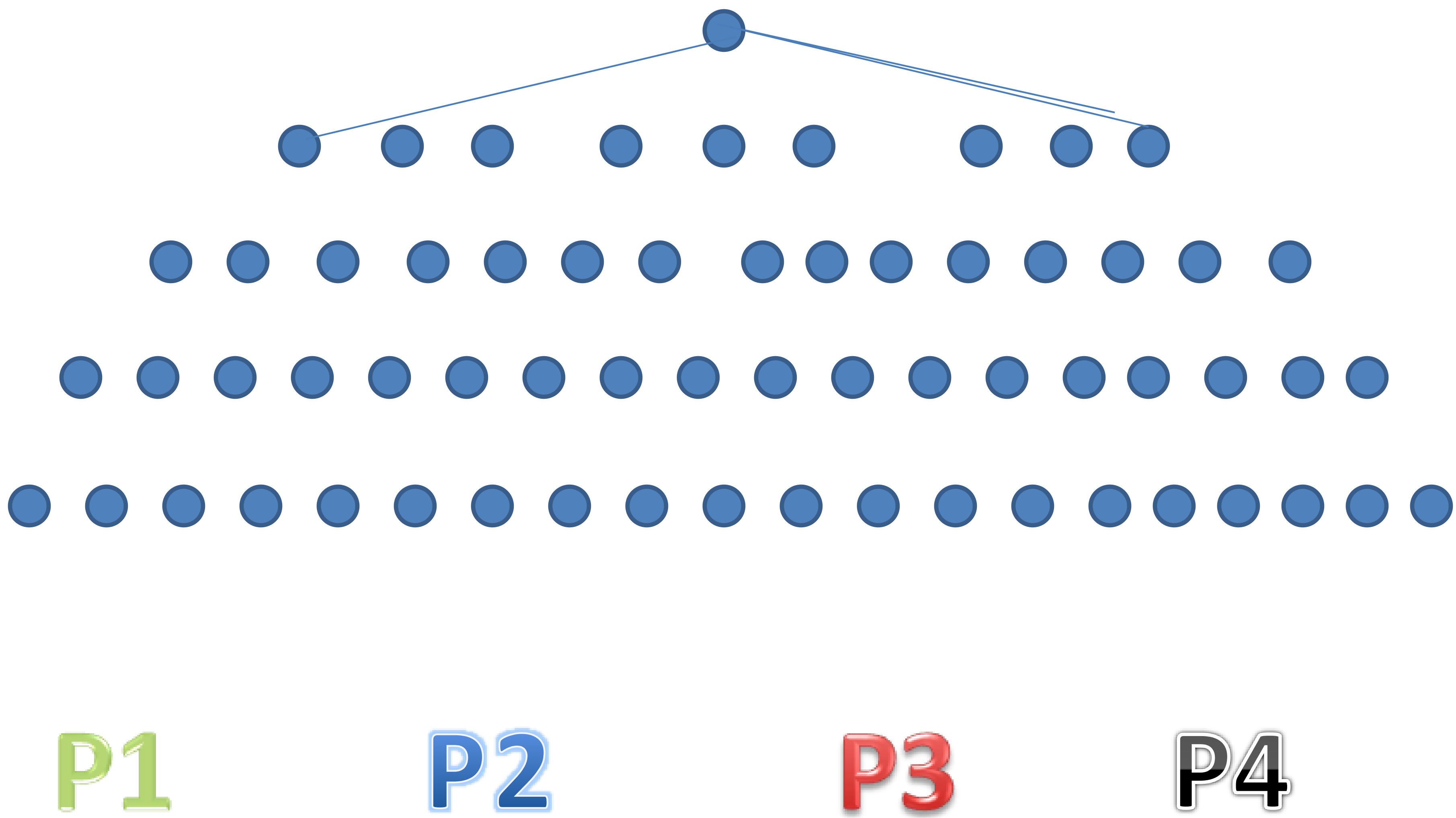
Conclusion

We are exploring how to build distributed systems that are attack-tolerant by design. The key idea is to implement systems that 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 will be 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 very high levels of abstraction by formally proving that designs are realizable. 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.

Illustrating Multiple Proofs



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.

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.

Development of Event Logic

Our Event Logic is an abstract account of distributed computing inspired by the work of Winskel and Plotkin in the 80's on Petri Nets. It spans from the very abstract notion of Event Class down to formal models of protocols that can be compiled to Message Automata and from them into code in languages such as Java, ML, Erlang, F#, and so forth.

Safety

We almost have a proof that our design at the level of event classes meets the requirements.

We also need to know property P2, that two decided values agree even if no processes fail.

Suppose that at some Pi the 2f+1 values collected in L are the same and likewise at Pj for j not i. Are the decided values equal?

Diversity at the Level of Proof

Multiple formal proofs can be “simultaneously” generated. We illustrate this by viewing a proof as a tree generated top down.

At the root of the upside down tree is a goal of the form |- G which asserts that proposition G is true (provable). Interior nodes have the form H |- G which means that from assumptions H the goal G can be proved.