Correct by Construction Attack-Tolerant Systems

Robert Constable

Mark Bickford

Robbert van Renesse

Cornell University and ATC-NY

DARPA Kickoff Meeting November 3, 2010

Talk Goals

Review our experience and capabilities in process synthesis and verification

Explain our approach to attack-tolerance

Outline ways we will contribute to CRASH

Outline

- Narrative thread – the story
- Event Logic and General Process Model
- Process Synthesis Methods
- Attack-tolerance

  approach to immunity and diversity

  example: consensus

The Story

The Cornell PRL group is known making it possible to use constructive proofs as programs and treat formal mathematics as a programming language. This has become a practical enterprise for certain applications.

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

The Story continued

We started by using IOA as our internal model of processes. 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 were forced to express the specifications more abstractly in order to complete the proofs and extract protocols.

The Story continued

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

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

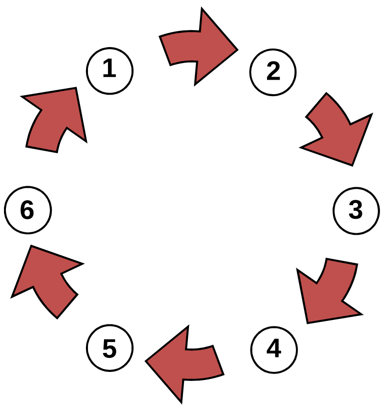
An Interesting Aside

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.

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 i to j

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

Specification, continued

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)

Realizing Leader Election

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

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

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

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

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

Lemmas

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

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

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

$(\mathtt{j} = \mathtt{ldr} \vee d(\mathtt{ldr}, \mathtt{j}) < d(\mathtt{ldr}, \mathtt{i}))$

By induction on causal order of `rcv` events.

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

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

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

but the right disjunct is impossible.

Finally, from property 4, it is enough to know

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

which follows from Lemma 1.

QED

Leader Election Message Automaton

state $me : \mathbb{N}$; initially $uid(i)$

state $done : B$; initially $false$

state $x : B$; initially $false$

action $vote$; precondition $\neg done$

effect $done := true$

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

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

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

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

action $leader$; precondition $x = true$

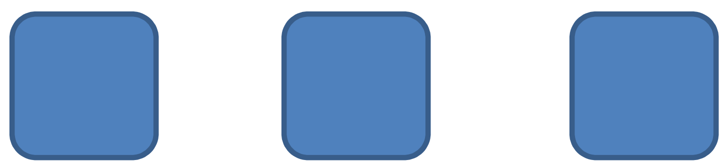
only $rcv_{in(i)}(vote)$ affects x

only $vote$ affects $done$

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

Consensus is a Motivating Example

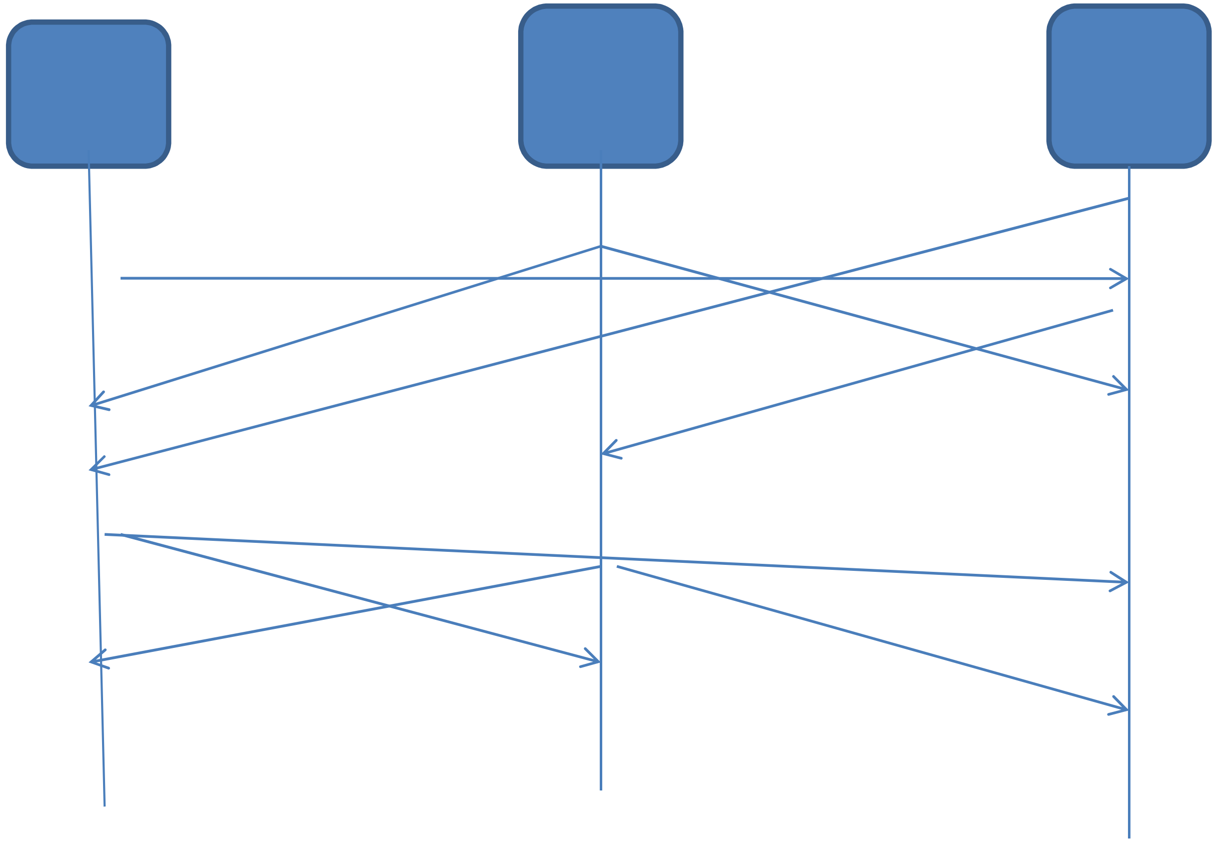
In modern distributed systems, e.g. the Google file system, clouds, etc., reliability against faults (crashes, attacks) is achieved by replication.



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

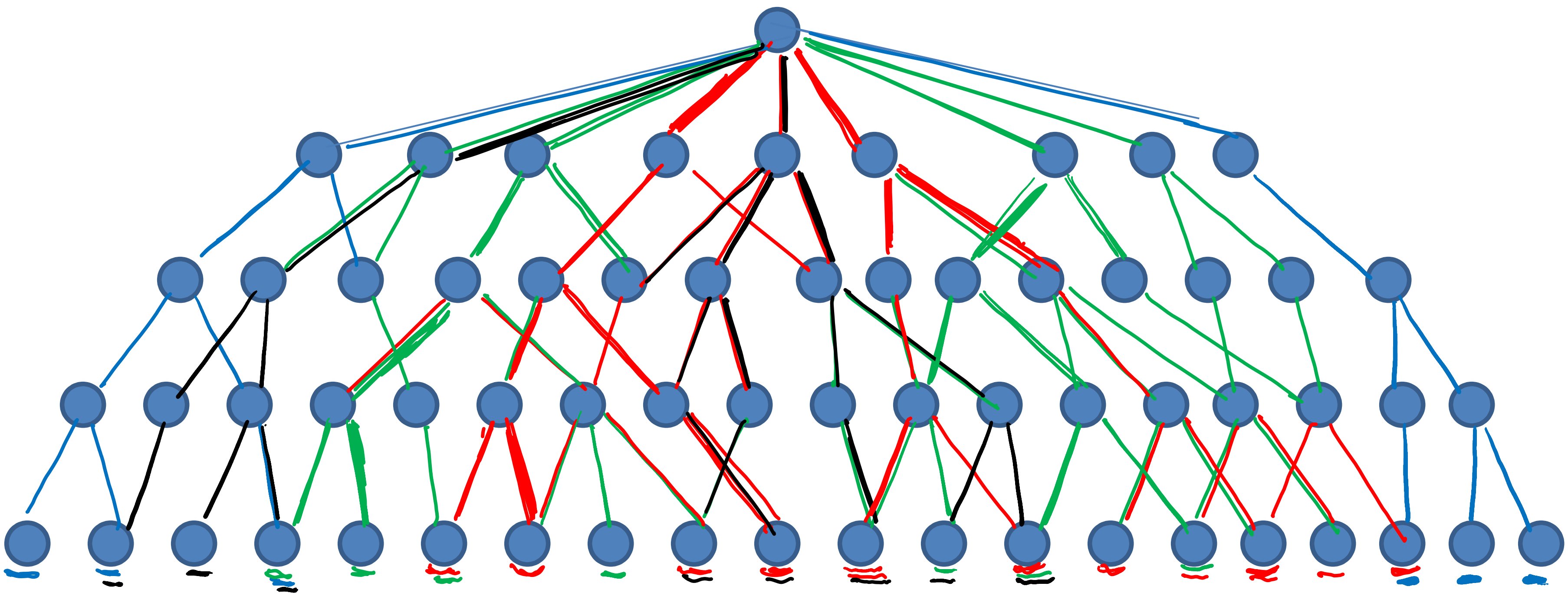
When we prove properties of a design, there are many options at several steps, and we are able to create multiple proofs at low additional cost. In the process we create new designs.

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.

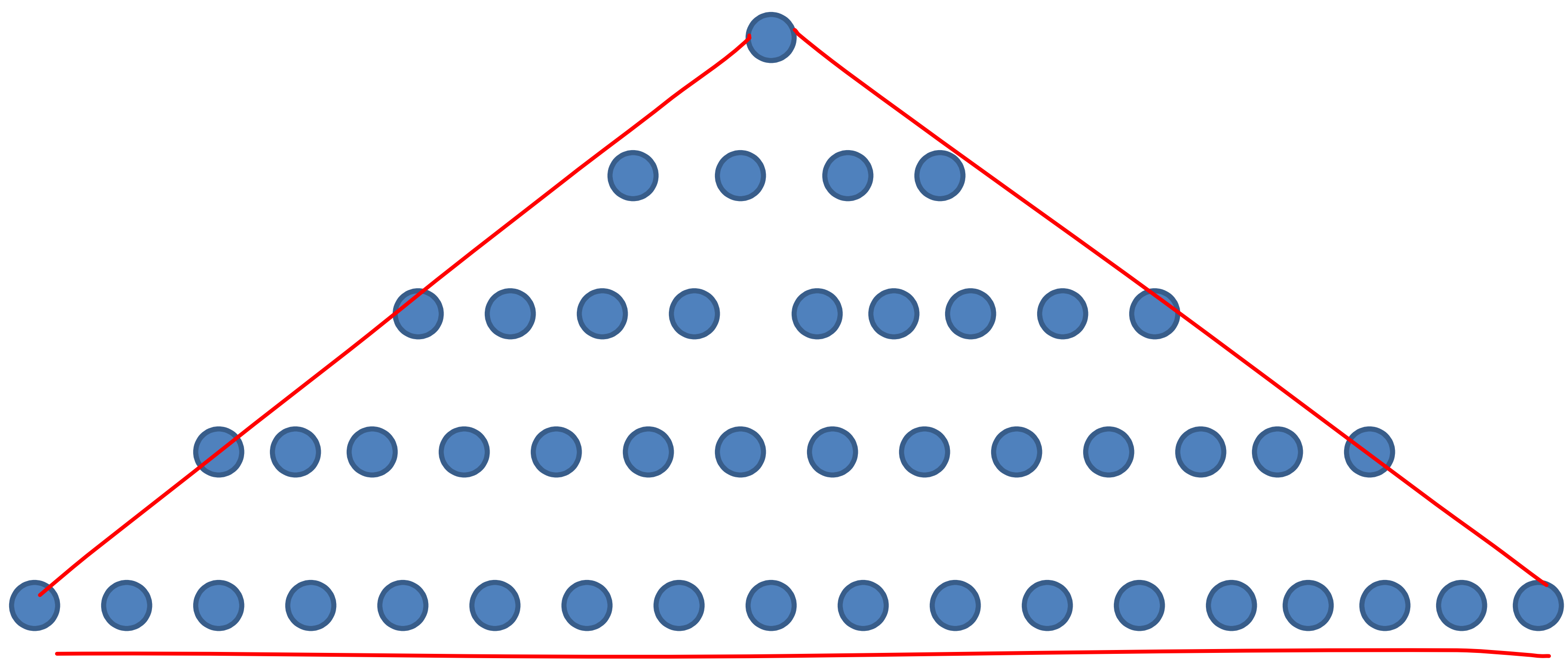
Diversity at the Level of Proof

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

Illustrating Multiple Proofs



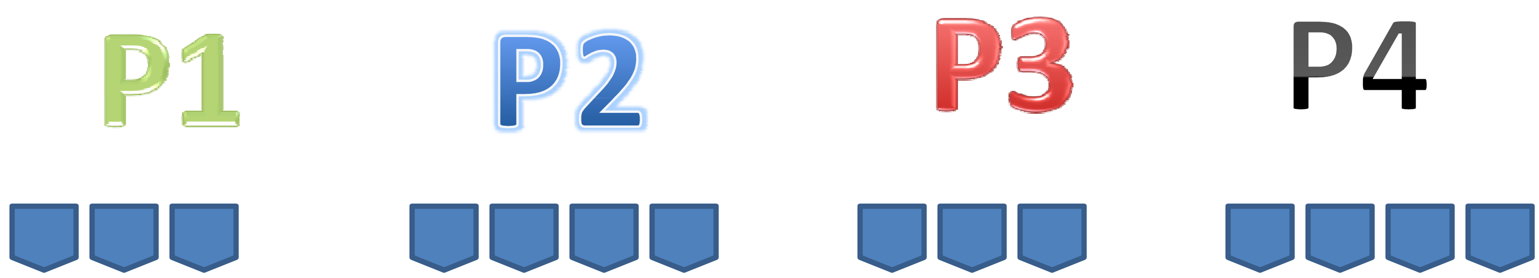
Illustrating Multiple Proofs



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.

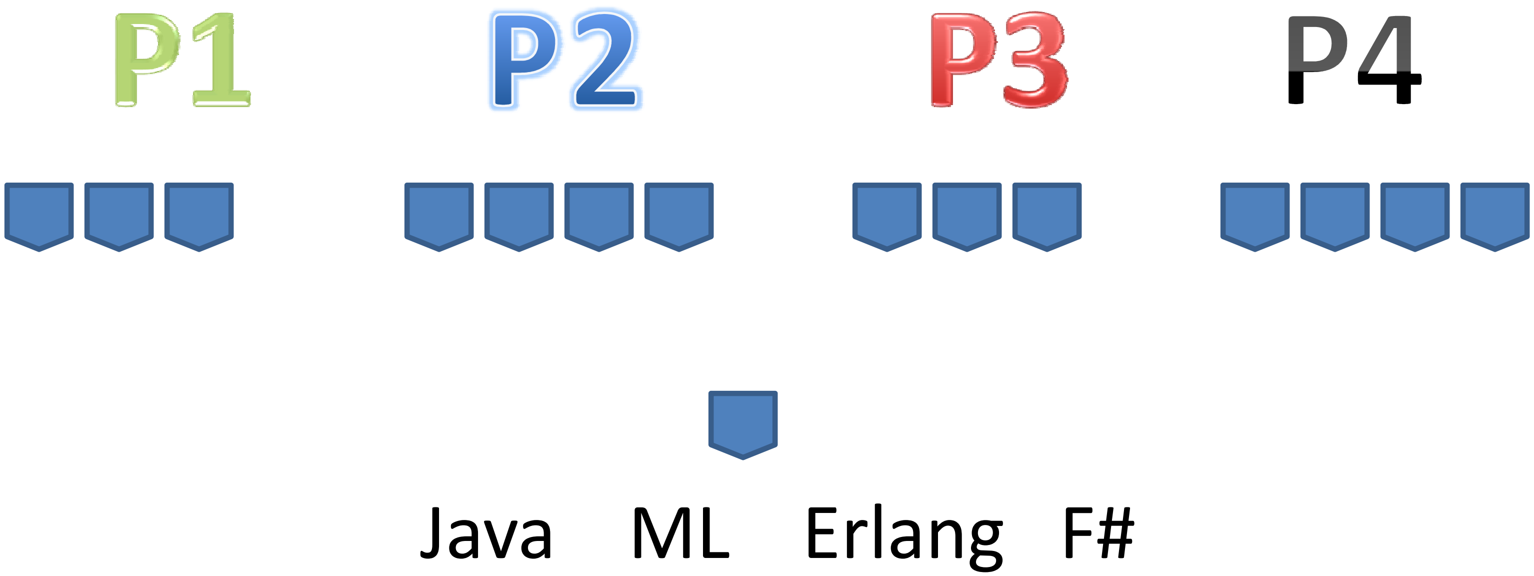


Programming Language Diversity

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

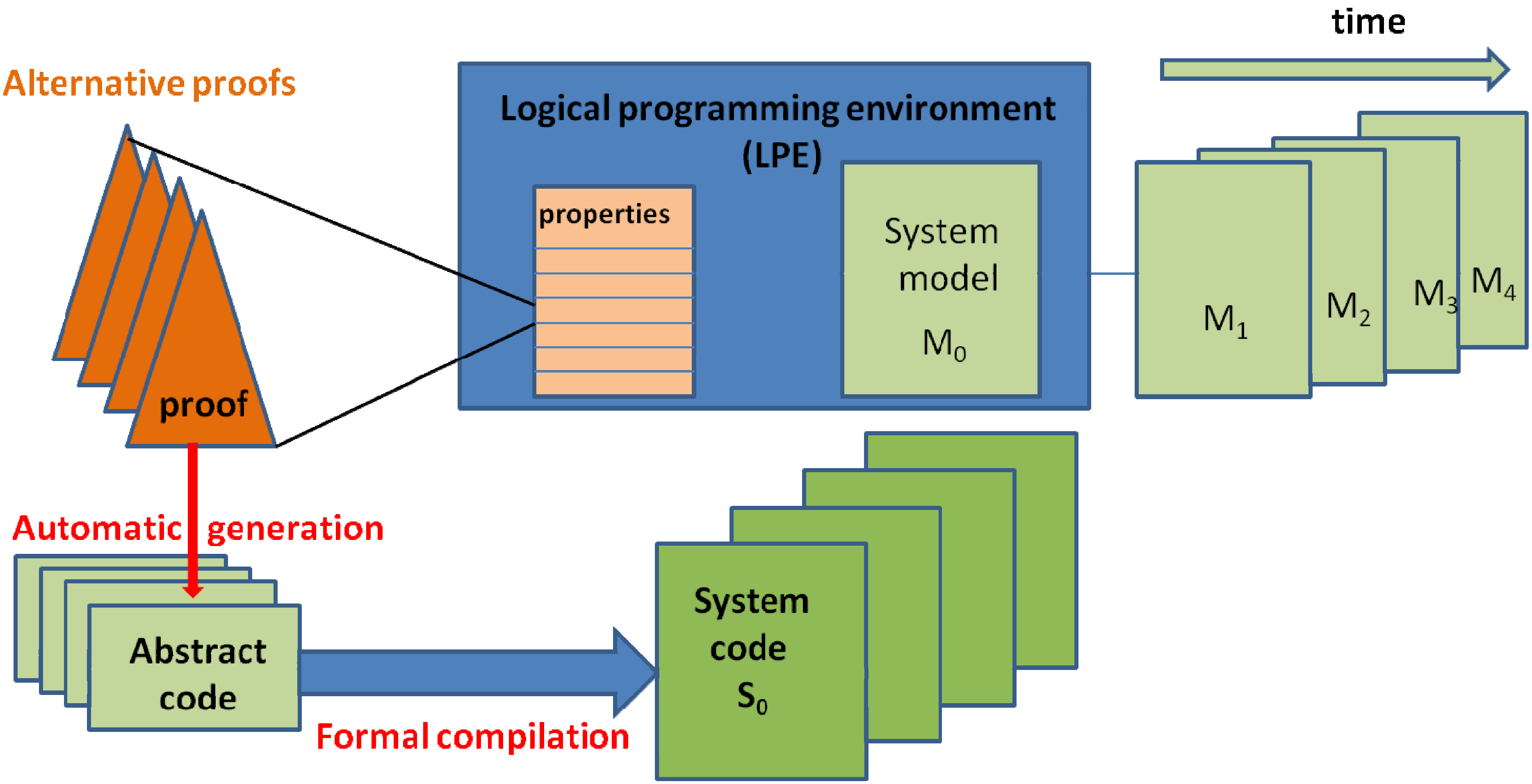
Combining all levels of diversity we are able to generate over 200 variants of a protocol in the best case.

Language Diversity



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

Design and construction of attack-tolerant systems

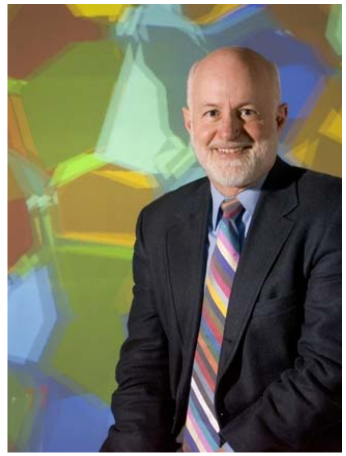


Correct-by-construction system code S_i semi-automatically evolves along with system models M_i

Correct by Construction Attack-Tolerant Systems

Robbert Van Renesse

Cornell University



Robert Constable

Cornell University

Mark Bickford

ATC-NY

Architecture Technology Corporation

Processes in Type Theory

- Process in Type Theory

  $\text{Process}(M,E) = \text{corec}(P.\ M[P] \longrightarrow P \times E[P])$

  - M represents messages (that can contain processes)
  - E represents the external effect (messages sent, …)
  - $\text{corec}(T.F[T]) = \bigcap n{:}\mathbb{N}.\ F^n[\text{Top}]$

- System = set of $\{\text{Loc} \times \text{Process}\}$

- Environment delivers messages & creates processes

Processes in Type Theory

- Process in Type Theory

  Process(M,E) = corec(P. M[P]$\longrightarrow$P $\times$E[P])

  - M represents messages (that can contain processes)
  - E represents the external effect (messages sent, …)
  - corec(T.F[T]) = $\bigcap$n:$\mathbb{N}$. F^n[Top]

- System = set of {Loc $\times$ Process}

- Environment delivers messages & creates processes

Processes in Type Theory

- Process in Type Theory

  $\text{Process}(M,E) = \text{corec}(P.\ M[P] \longrightarrow P \times E[P])$

  - M represents messages (that can contain processes)
  - E represents the external effect (messages sent, …)
  - $\text{corec}(T.F[T]) = \bigcap n{:}\mathbb{N}.\ F^n[\text{Top}]$

- System = set of $\{\text{Loc} \times \text{Process}\}$

- Environment delivers messages & creates processes

Processes in Type Theory

- Process in Type Theory

  $\text{Process}(M,E) = \text{corec}(P.\ M[P] \longrightarrow P \times E[P])$

  - M represents messages (that can contain processes)
  - E represents the external effect (messages sent, …)
  - $\text{corec}(T.F[T]) = \bigcap n{:}\mathbb{N}.\ F^n[\text{Top}]$

- System = set of $\{\text{Loc} \times \text{Process}\}$

- Environment delivers messages & creates processes