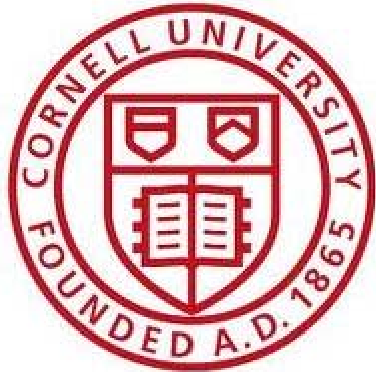
Correct by Construction Attack-Tolerant Systems



Robert Constable

Cornell University

Mark Bickford

ATC-NY

Robbert Van Renesse

Cornell University

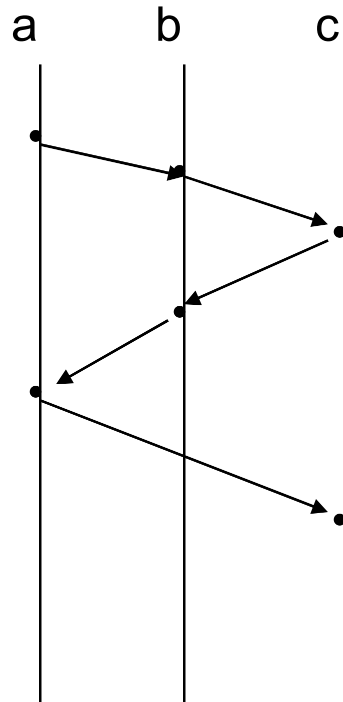
Correct by Construction Systems

- Refine high-level specifications to code
  - Refinement process produces a correctness proof
- A mature discipline for functional programs
  - "proofs as programs"
- For systems of concurrent agents generalize to "proofs as processes"
  - Modulo assumptions on operating environment

Event Logic

Event ordering (EO)

⟨E, loc, ≤, info⟩



- Events
  - have location (loc) and value (info)
  - are causally ordered (≤)
- Causal order
  - is well-founded
  - is locally finite
  - totally orders events at each location
- Implemented in Nuprl
  - Over 3,000 theorems
  - Easily portable to Coq, Isabelle, etc.

Event Classes

- An *event class* X of type T
  - recognizes a set of events, E(X)
  - assigns each e in E(X) a value X(e) : T
- It abstractly represents an interface
- Consensus interface:

  Event classes, Input and Decide, such that

Validity $\forall\, e_2\text{:E(Decide)}.\ \exists\, e_1\text{:E(Input)}.$

$$e_1 \leq e_2 \ \&\ \text{Decide}(e_2) = \text{Input}(e_1)$$

Agreement $\forall\, e_1,e_2\text{:E(Decide)}.\ \text{Decide}(e_1) = \text{Decide}(e_2)$

Propagation Rules/Constraints

- Specify message propagation by

$$X =^{f}\Rightarrow Y \text{ @ } g$$

Each X-event with value v causes
Y-event(s) with value f(v)
at each location in list g(v)

- Constrain message propagation by

$$X \Leftarrow^{f}\Rightarrow Y \text{ @ } g$$

… and each Y-event is caused by an X-event

- Systems of concurrent agents can be described abstractly as a set of event classes, propagation rules, and propagation constraints

Example: Paxos

- Refine Agreement: introduce ballots & quorums to define spec Agreement$_1$:

$\exists$Ballot:Class($\mathbb{N}\times$T). $\exists$W:{2-Quorum-sys (on locs)} .

$\forall\alpha,\beta$:E(Ballot). Ballot(α)$_1$=Ballot(β)$_1$ $\Rightarrow$

Ballot(α) =Ballot(β)

& $\forall$v:T. $\forall$b:$\mathbb{N}$. $\forall$Q $\in$ W.

($\forall$a $\in$ Q. $\exists$ α :E(Ballot). Ballot(α) = $\langle$b,v$\rangle$) $\Rightarrow$

$\forall\beta$:E(Ballot). b < Ballot(β)$_1$ $\Rightarrow$ Ballot(β)$_2$=v

& $\forall$ β :E(Decide). $\exists$Q $\in$ W. $\exists$ b:$\mathbb{N}$. $\forall$a $\in$ Q. $\exists$ α :E(Ballot). Ballot(α) =$\langle$b,Decide(β) $\rangle$

- Prove Agreement$_1$ $\Rightarrow$ Agreement

Example: Authentication Protocols

- Security Event Logic:
  - Models "unguessable" values (nonces, signatures, ciphertexts, private keys) as members of type *Atom*
  - Extends event logic with event classes {*New, Sign, Verify, Encrypt, Decrypt, Snd, Rcv*} and axioms
  - Defines Authentication Protocol as in PCL (Datta, Mitchell, et al) using "matching conversation".
  - Totally automates (with tactics) proofs of protocols such as "Challenge Response"

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 {Loc ✕ Process}

- Environment delivers messages & creates processes

Proofs as Processes

- S: System + Env:Environment defines
  - --A Run, R = Run(S,Env)
  - --Events in R are the delivered messages
    - They have location, info, and causal order
    - A run R defines an event ordering EO(R)
  - --S *realizes* ψ *assuming* φ :

    $\forall env \vdash \varphi\ .\ EO(Run(S,env)) \vdash \psi$
  - --S *strongly realizes* ψ :

    $\forall S'\ .\ S \subseteq S' \Rightarrow S'$ realizes ψ
- From a constructive proof that ψ is realizable we can generate an implementation of ψ

Process Model (Version 1.0)

- Environment provides
  - Message passing
  - Process Creation

- Correct compiler E# to System
  - E# programming language defines
    - basic event classes
    - classes that are combinations of others
    - propagation rules and constraints

  - E# compiler is

    Correct-by-construction
    Produces ⟨tcc, spec, system⟩ such that system realizes spec , provided the type checking constraint tcc is true

Process Model (Version 2.0)

- Introduce a model of shared memory
  - generalize processes, propagations, constraints

- Characterize the assumptions on the environment needed to create strong realizers
  - E.g. access control and authentication primitives

- Relate these to the OS services provided by CRASH prototypes

Synthetic Diversity

- Diversity provides raw material for adaptation to survive cyber-attack (e.g. a moving target)

- It should be introduced at different levels of abstraction

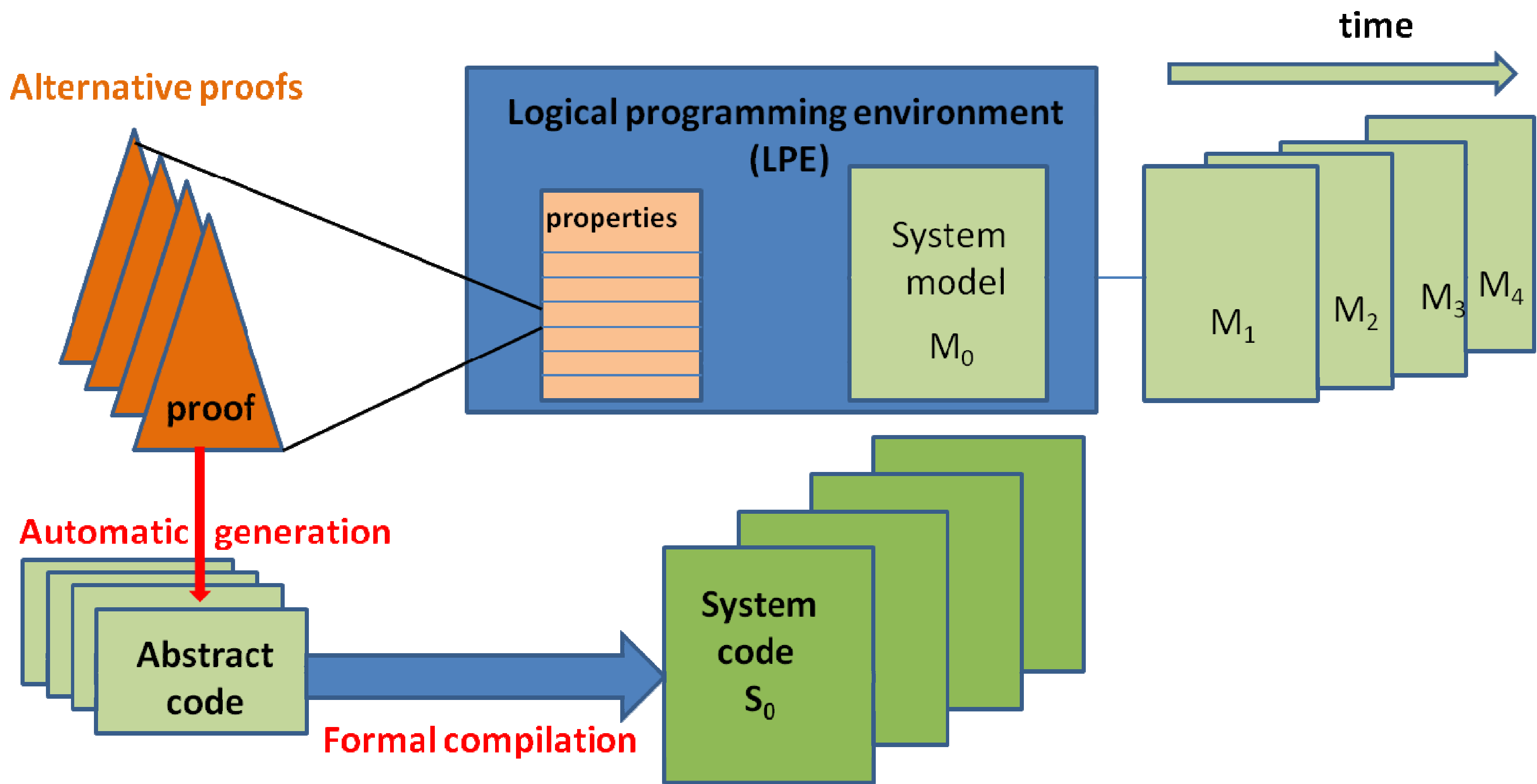
  Correct-by-construction method can introduce diversity at *high levels* of abstraction from variant proofs

  - Theory modification tools provide efficient ways to generate proof variants.

  Process code can be generated in different languages (Java, Erlang, C++, etc)

Coping with Change

- CRASH evolution will require us to quickly change our model
- Logical Programming Environment (LPE) provides tools for managing change
- Database of structured information
  - Definitions, Theorems, Tactics, Update objects, Display forms
- Theory modification tools
  - Transformations, replay/rebuild, proof modification heuristics


Design and construction of attack-tolerant systems
time
Alternative proofs
Logical programming environment (LPE)
properties
System model
M_0
M_1
M_2
M_3
M_4
proof
Automatic generation
Abstract code
Formal compilation
System code
S_0
Correct-by-construction system code S_i semi-automatically evolves along with system models M_i