Modeling Paradigms & Tools for Autonomy Management in Complex Networked Systems:
A case for Probabilistic Model-Checking based approach

Kaliappa Ravindran (Professor) City University of New York (CUNY -- Graduate Center)

Graduate student participants:

Gregory Javens (Ph.D.)

Gwang Yu Ding, M.S. [currently employed in China]

July 2026

Salient points of our research

1. Self-managing systems with cyber-physical components

2. Algorithm verification *versus* system assessment

   (non-functional properties:
   e.g., stability, convergence, agility, adaptivity, etc)

3. Software engineering view of adaptive networked systems

4. Model-checker tools for management of adaptation processes
   (Advantage: *Reusable, Reconfigurable, Higher quality of adaptation*
   Need for rapid prototyping is exacerbated as systems become complex)

5. ***System-level Case Studies*:**
   - i) Resilient supply-chain Management for Product Distributions
   - ii) UAV-based surveillance to Enforce Secure Cyber-fencing
   - iii) Collaborative driving autonomous cars in intelligent highways

6. Discussion of alternate modeling tools:
   - 6.1 Probabilistic Model-checkers (PROMELA*, PRISM, STORM, PAT, . .)
   - 6.2 Operational models (closed-form mathematical expressions, . .)
   - 6.3 Miniature prototypes of real-world systems

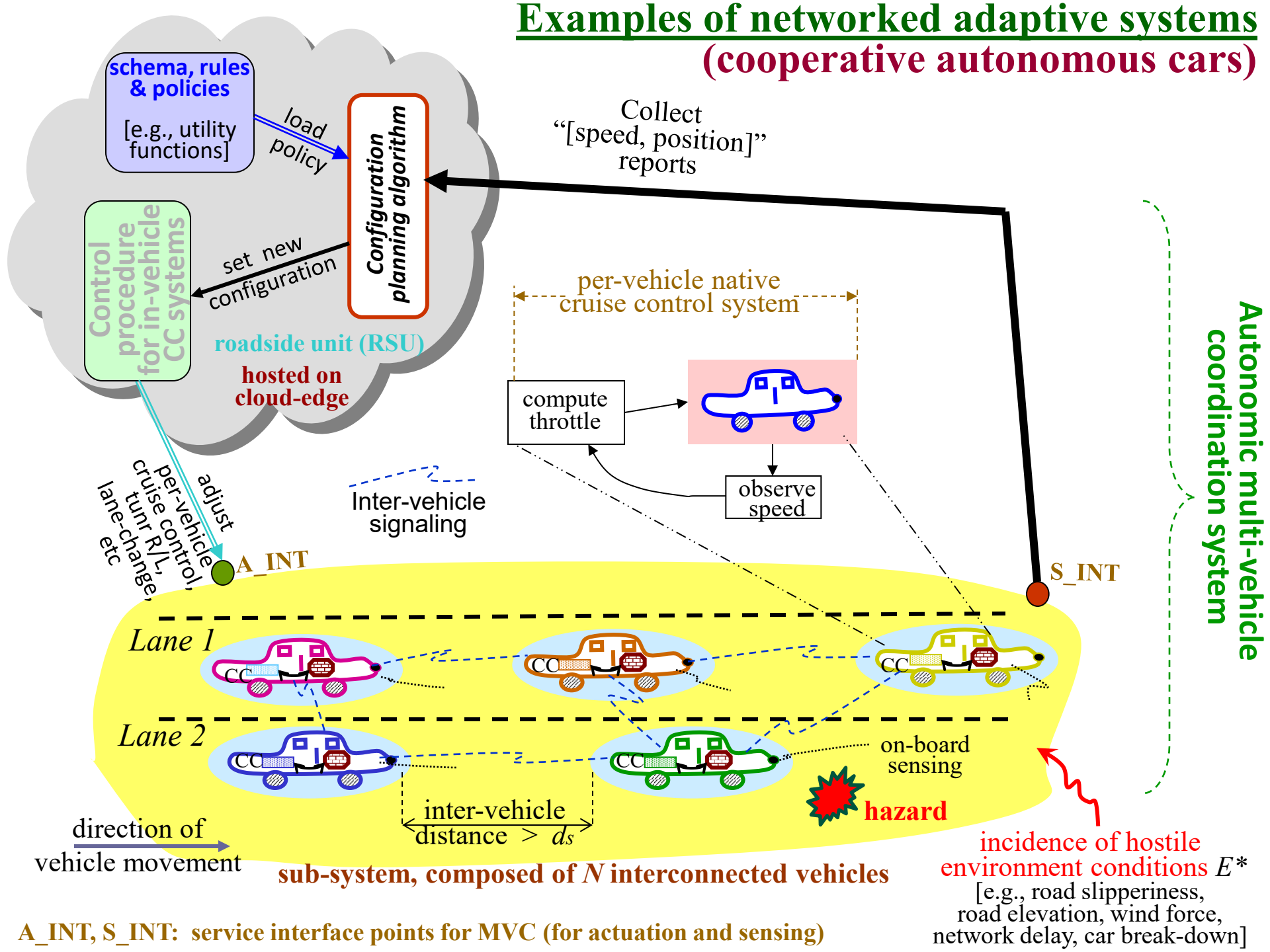
Fundamental concepts of complex network systems

--- How complex is the handling of "complexity" ??

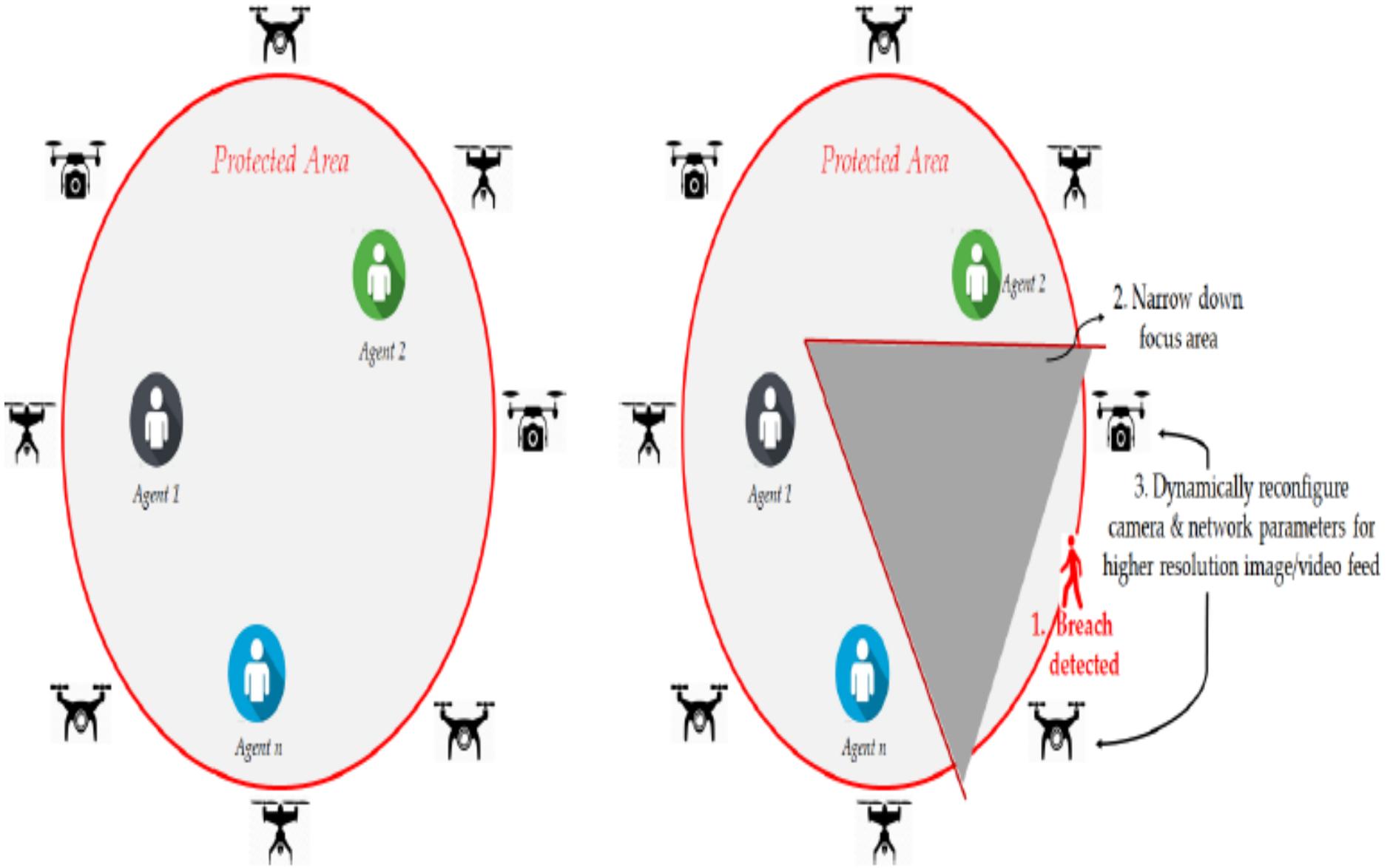
--- Proactive planning to handle faults (instead of reacting)

[balancing between pessimistic *vs* optimistic approaches]

Examples of networked adaptive systems (cooperative autonomous cars)



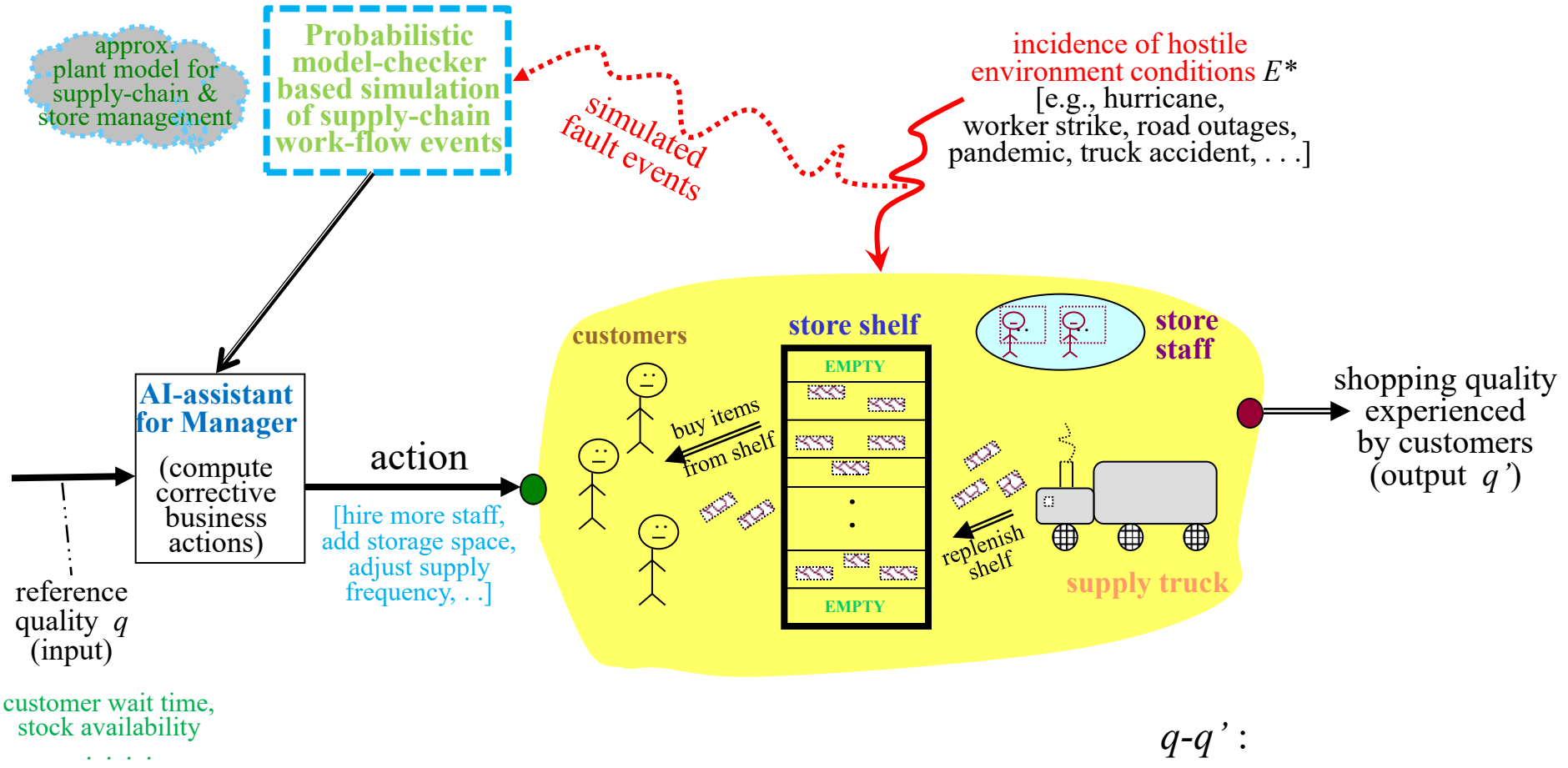
Example 2: *Cyber-fencing of Metro-cities by multiple UAVs or drones*



(a) UAV Swarm Setup

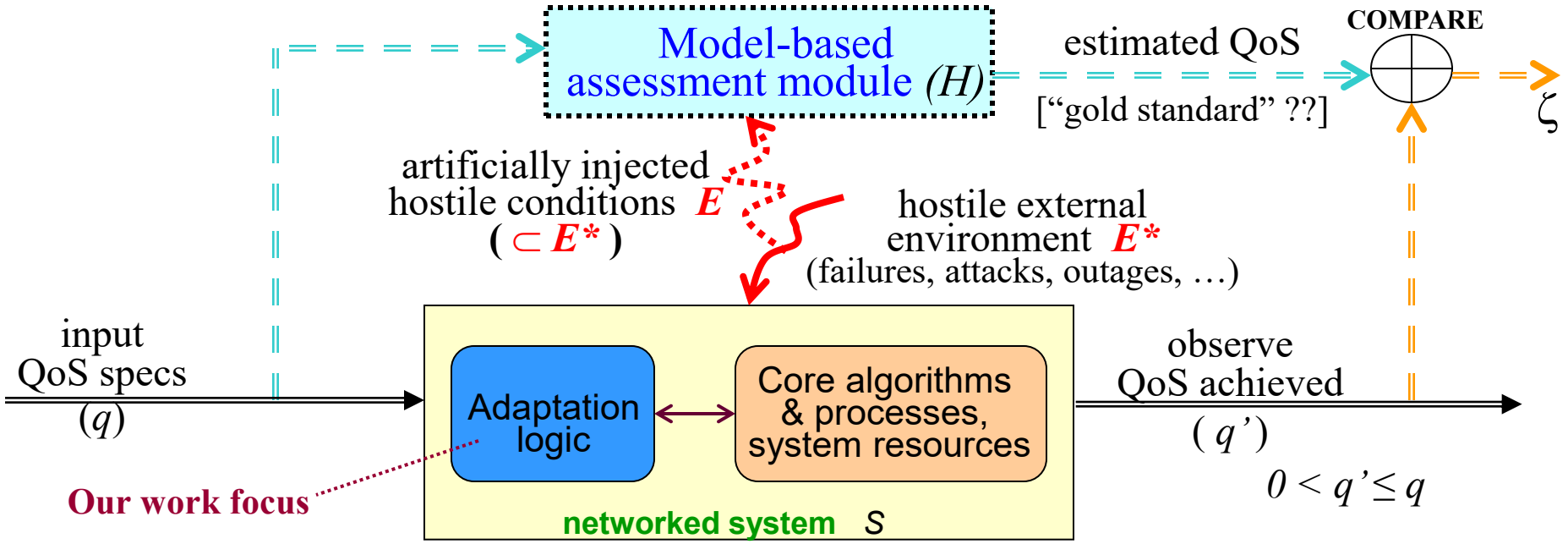
(b) Intrusion Detection

Example 3: Fault-resilient supply-chain network for product distribution



q-q' :
tracking error in the shopping quality, arising from managerial decision-making process

Quantitative view of designing of resilient systems (digital-twin)



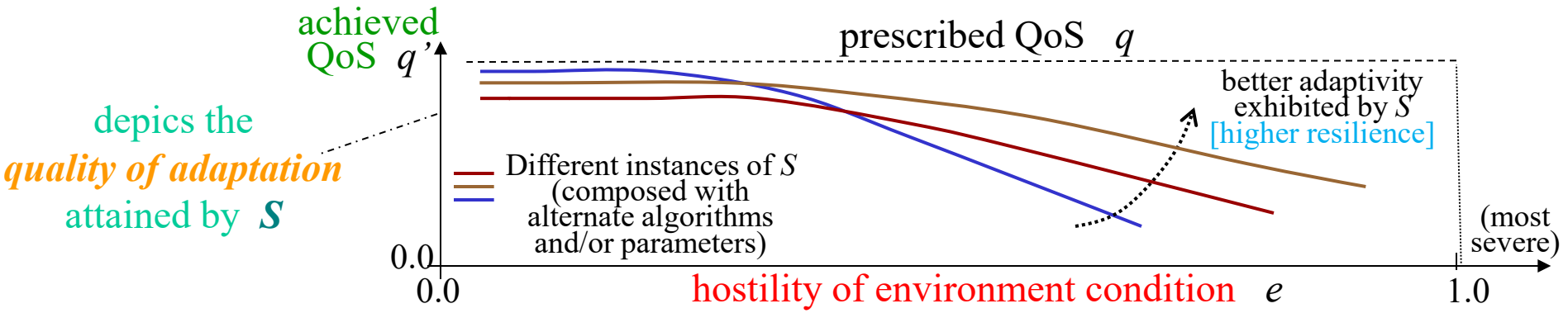
Ψ = 1.0 ➔ 100% trustworthiness bestowed on S for meeting the QoS specs

= 0.0 ➔ not trustworthy at all

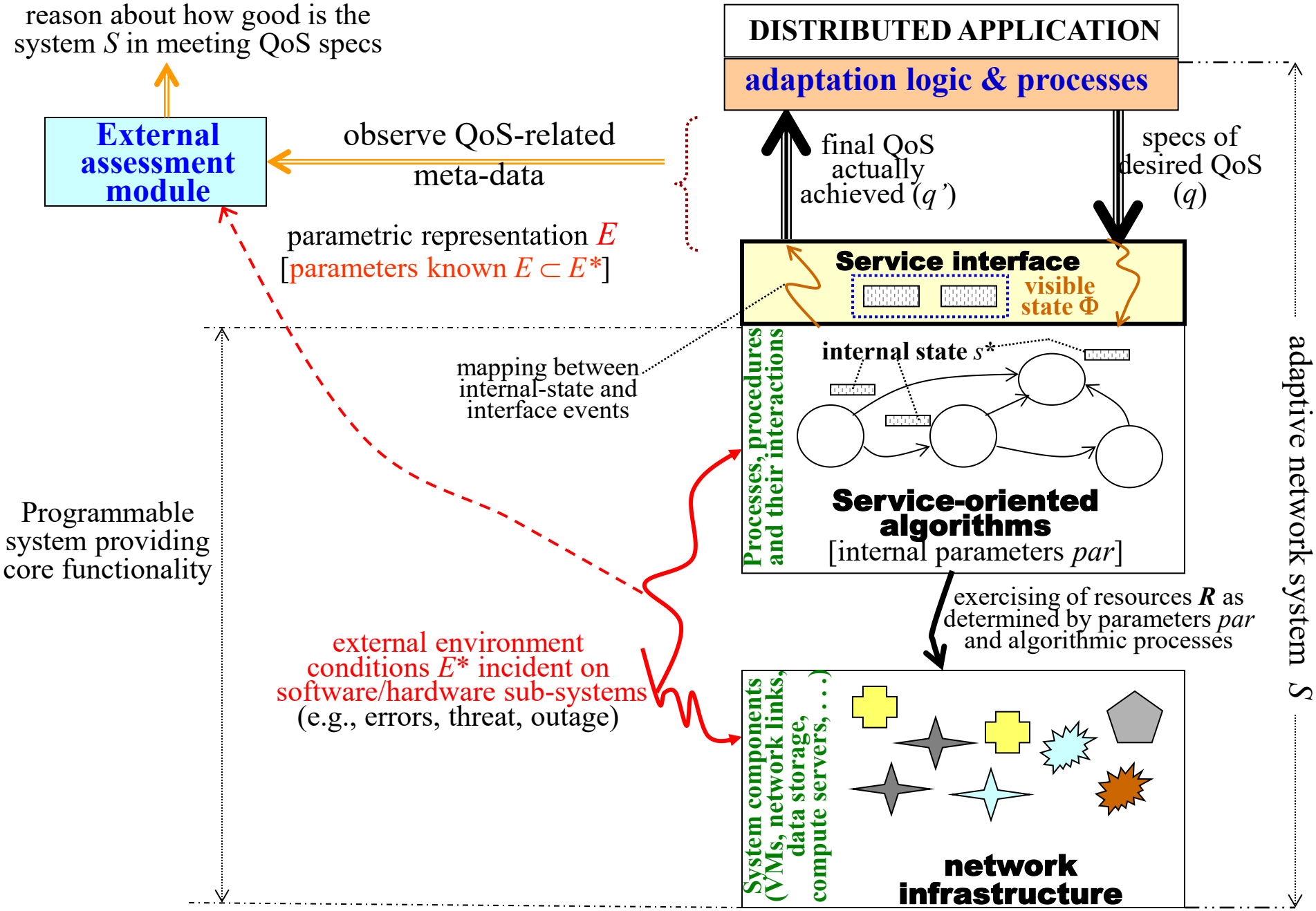
A coarse measure of the "goodness" of system S under harsh condition $e \in E^*$:

$$\Psi_{(S,q,e)} = [1 - \frac{(q-q')}{q}]$$

$$0.0 < \Psi_{(S,q,e)} \leq 1.0$$



Software Engineering view of self-managing network systems



Motto behind resilience assessment

A system that is good
but is not *verifiably good* is
not good enough !!

[e.g., quality rating of restaurants, hotels, taxi services, . . .]

3-star, 4-star, etc

In contrast, **system auditing** reasons about
why an actual QoS attained is less than a desired QoS specs
[say, less resources, node outages, laxity of algorithms, . .]

Uncertainties emanating from external environment

QoS specs q, algorithm parameters par,
system resource allocation R are usually controllable inputs

In contrast, environment parameters $e \in E^*$ are often uncontrollable and/or unobservable, but they do impact the system-level performance (e.g., component failures, network traffic fluctuations, attacks, etc)

environment parameter space:

$$E^* = E(\text{yk}) \cup E(\text{nk}) \cup E(\text{ck}) \cup E(\text{uu})$$

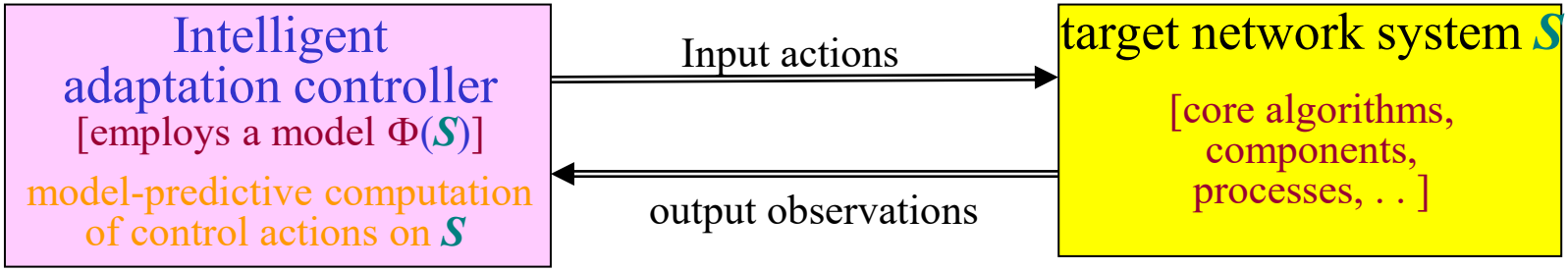
- $E(\text{yk})$: parameters that the designer knows about (known "knowns")
- $E(\text{nk})$: parameters that the designer does not currently know about (knowable "unknowns")
- $E(\text{ck})$: parameters that the designer can never know about (known "impossibilities")

What about unknown "unknowns" $E(\text{uu})$?? → (late) Hon. D. Rumsfeld, US defense secretary

Algorithm design decisions face these uncertainties --- so, designer makes certain assumptions about the environment
[e.g., at most 3 UAVs in a 10-vehicle surveillance team will fail during operations].
When assumptions get violated, say, due to severe attacks, algorithms fall short of what they are designed to achieve
→ Evaluate how good an algorithm performs under strenuous environments

Categorization of modeling paradigms

Model $\Phi(S)$ predicts the future behavior of a networked system S



1. Operational models of S

   Quickly computable, but low accuracy of prediction

   (due to the simplifying assumptions of **piecewise-linearity, function-separability, . .**)

2. Close-to-real software prototyping of S **(to mimic its run-time behavior)**

   High accuracy of prediction, but building system prototype incurs lot of development efforts, and long turn-around time

3. Model-checker based discrete-event simulation of S

   Offers a middle ground: **Reasonable accuracy of prediction, Rapid & flexible composition of alternate system-level functions, model reusability & verifiability, low model-development costs**

Metaphorical statement of Prof. Judea Pearl (UCLA) on Modeling Accuracy

[well-known for works on Game-Theory & Epistemic reasoning]

The best model of a "cat" is the "cat" itself !!

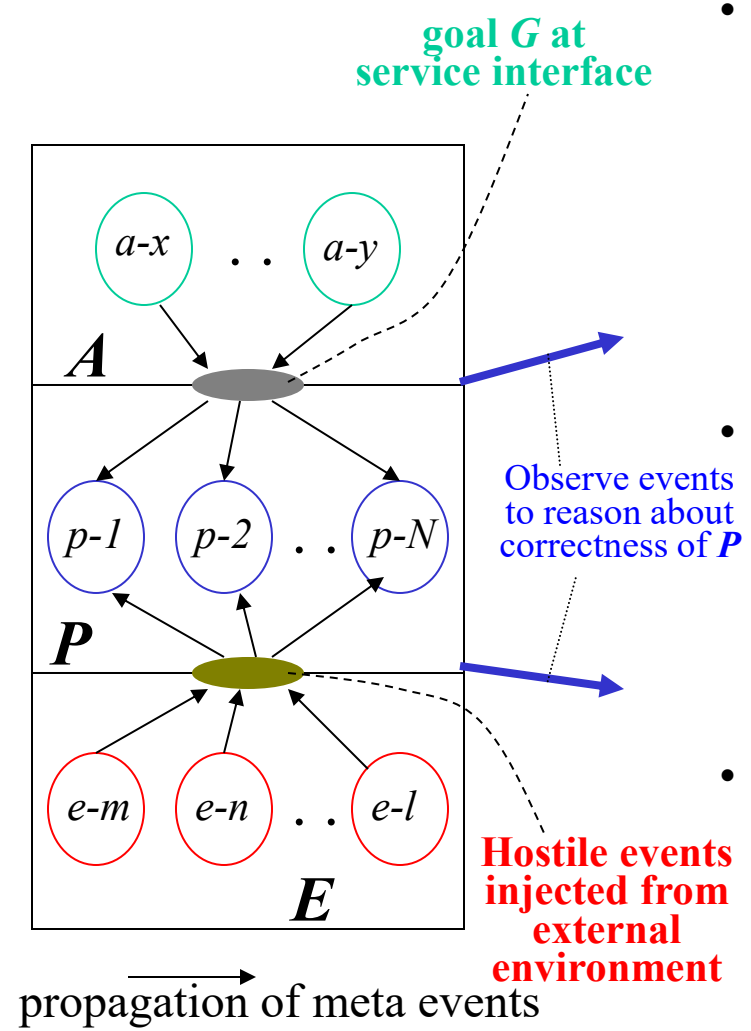
Any computationally-tractable "modeling paradigm" is bound to have inaccuracies (due to the simplifying assumptions made about the system)
[it is just a question how inaccurate one paradigm is relative to others]

➔ **What are the sources of inaccuracies in a modeling paradigm**

➔ **What is the "gold-standard" for modeling accuracy ??**

Event-based view of *Distributed algorithm* in a networked system of things

Algorithm ***P*** : governs the interactions between two or more processes through a set of rules to realize a service-layer property *G*



- externally visible interface to ***P***, seen by application ***A***, that prescribes *safety, liveness* conditions to be met by ***P***

  safety → "bad thing" never happens during execution of ***P***

  liveness → "good thing" thing eventually happens

- Algorithm processes of ***P*** propagate meta-events & actions to counter the effects of hostile environment ***E*** (e.g., `packet retransmit' action to counter `packet loss')

- operating environment ***E*** that attempts to disrupt the pursuit of ***P*** towards goal *G* (e.g., `packet loss' during `data transfer' over a network)

What does the correctness of a distributed algorithm mean ??

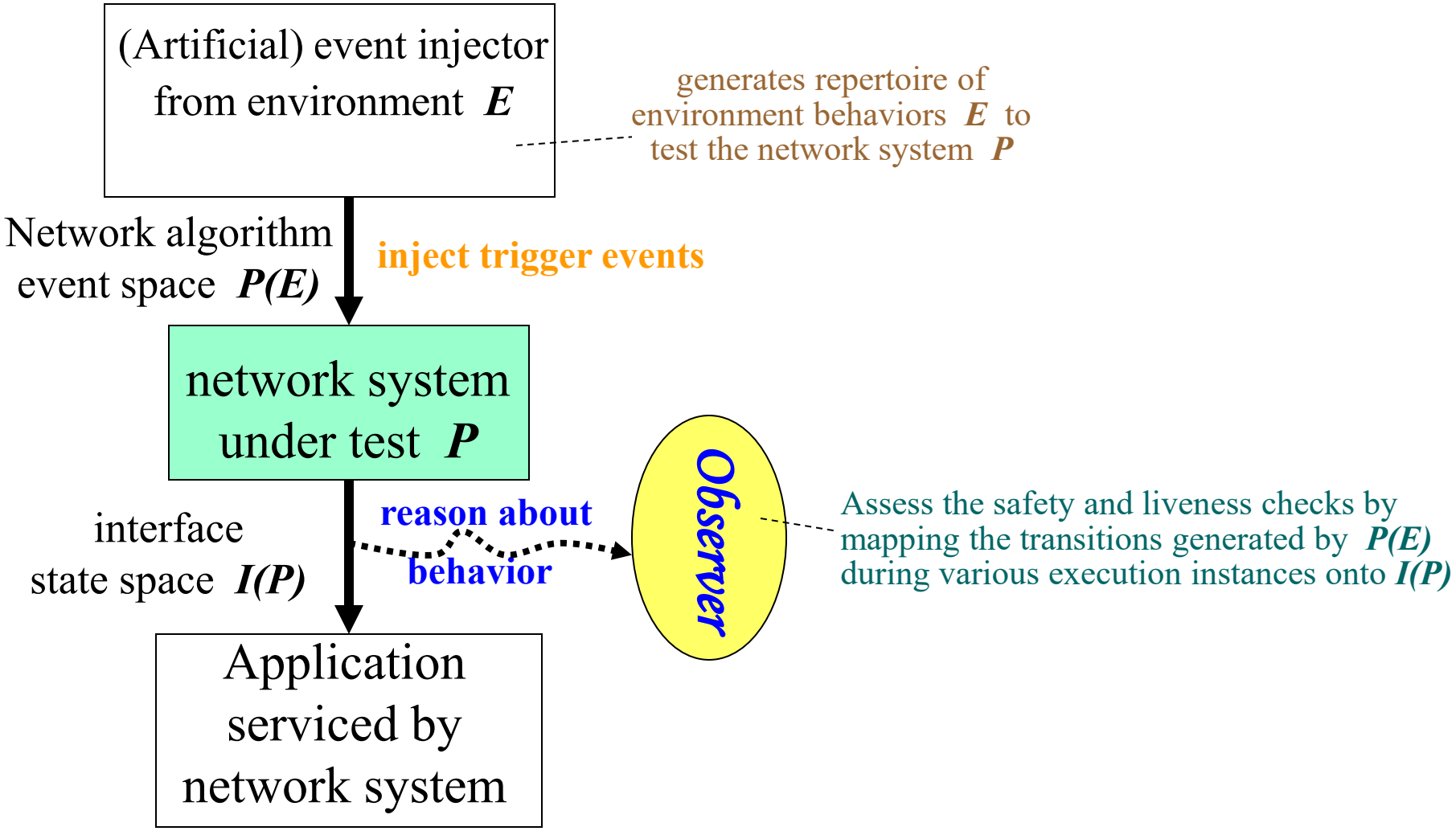
A distributed algorithm ***P*** implements a set of functions, often running on different machines on a network, to meet the goal ***G*** set by applications {***A***}, in the presence of perturbations emanating from the operating environment ***E***

E.g., 'Uber taxi driver' interacts with multiple entities/data (booking site, weather report, road transport agency, etc) for on-time passenger drop-off

CLAIM:
P(E) meets *G* under certain assumptions about ***E***

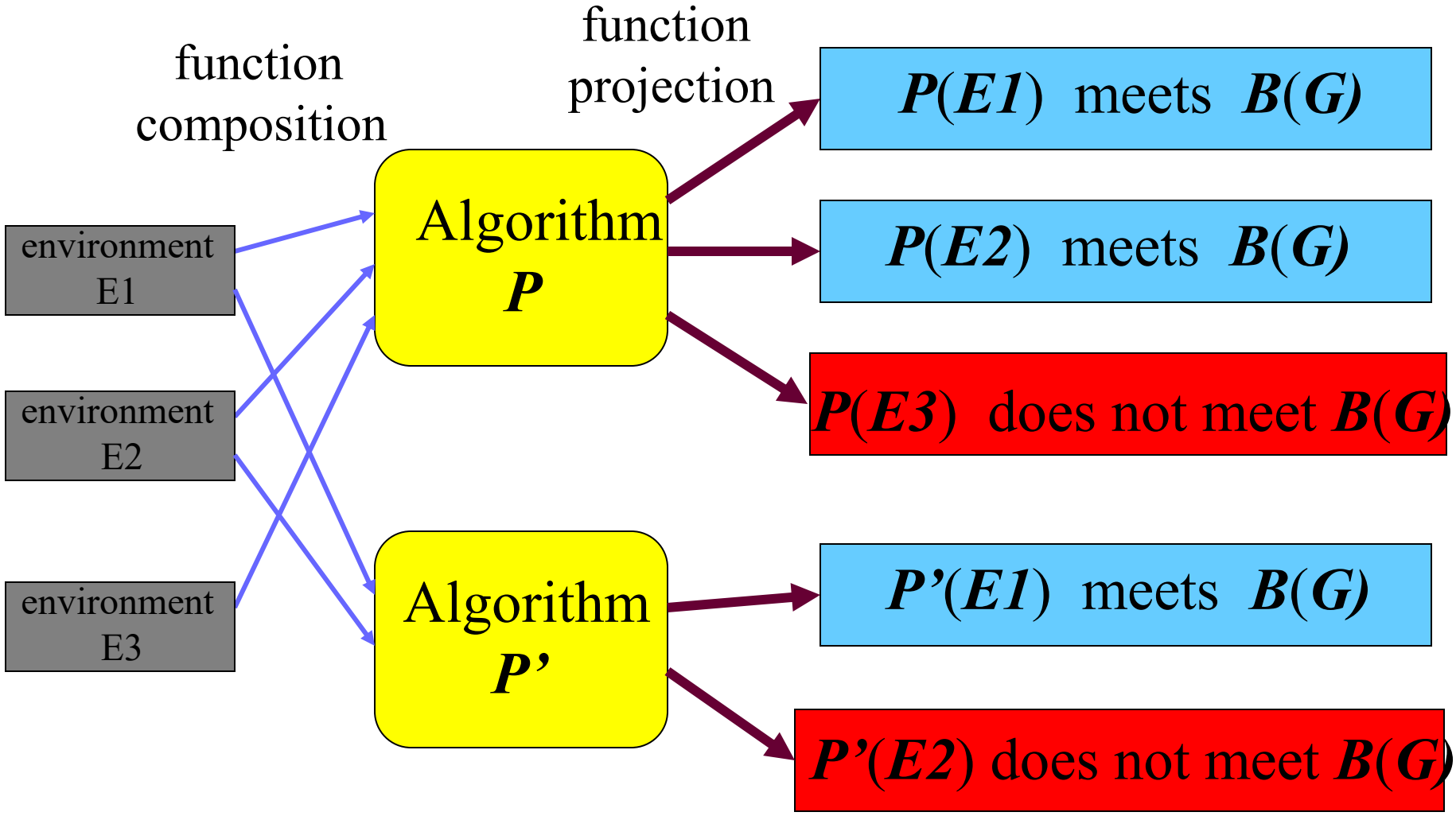
Algorithm correctness refers to a verification of the above claim, i.e., checking through some means as to whether ***P*** does in fact meet the goals ***G*** when subject to the disturbances emanating from ***E***

Testing modules for fault-tolerance & resilience



Employ *state transition projection relations* for ***automating*** the safety and liveness checks from the execution traces of state transition sequences generated by ***P(E)*** at run-time

‘function-based’ view of algorithm verification



P’ is less resilient than ***P*** (so switch from ***P’*** to ***P*** if/when needed)

E3 is more hostile than ***E2;*** ***E2*** is more hostile than ***E1.***

B*(*G*)**: Assertions on the service interface state depicting goal ***G

Quick tour of PROMELA language for model-checking

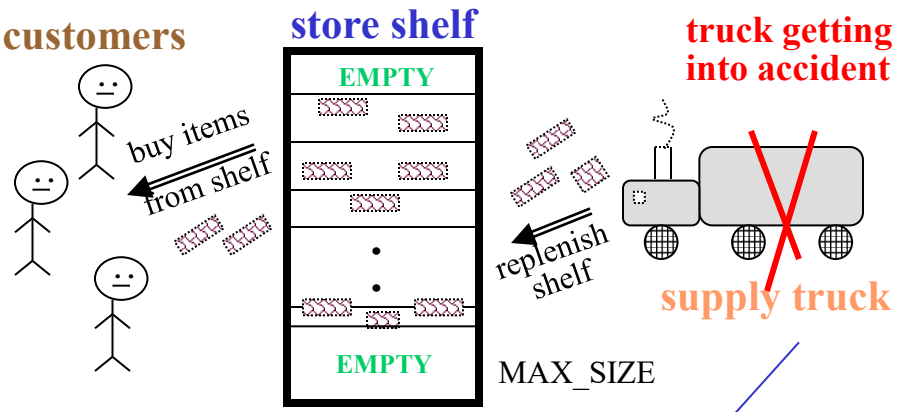
Our extensions to PROMELA (Protocol Modeling Meta-Language)
[originally designed at AT&T Bell-Labs, in mid-90's (to analyze telecom systems)]

Key conceptual elements in formulating model **Φ(S)** of a network system ***S***:
--- Cause-and-effect relationship between actions/events
--- State-based encapsulation of past and future events/actions

- PROMELA software overlays to assign state-transition probabilities and SPIN trace-analysis (with home-grown extensions to explicitly control the entry of PROMELA threads for the execution of ***Guard*** statement blocks)
- Rapid composition of algorithm functions into PROMELA processes
- Discrete-event simulation of message flows & process states and explicit causality tracking (use randomly-generated events to trigger the simulation)
- Consideration of other probabilistic modeling languages/tools: such as PRISM (Oxford Univ), STORM (Tech Univ. of Aachen), FSQ (Carnegie-Mellon Univ), and PAT (Natl. Univ. of Singapore)

Sample PROMELA-modeling of store supply-chain

```
PROMELA-proc customer(x)
    spont(x) → send(purchase_rqst, ch_shelf)
    receive(x, ch_shelf) → {if (x == product)
                                my_state := HAPPY;
                                pay at check-out counter;
                            if (x == OUT_OF_STOCK)
                                my_state := ¬ HAPPY;
                                send(COMPLAINT,ch_shelf);}
                :
                :
PROMELA-proc  store_shelf()
    receive(supply, ch_truck) → {shelf_full →
                                    send(NO_SPACE, ch_truck);
                                 ¬shelf_full →
                                    if (inventory+supply ≥ MAX_SIZE)
                                        shelf_full := true;
                                        inventory := MAX_SIZE;
                                    else
                                        inventory =+ supply;}
                :
                :
```

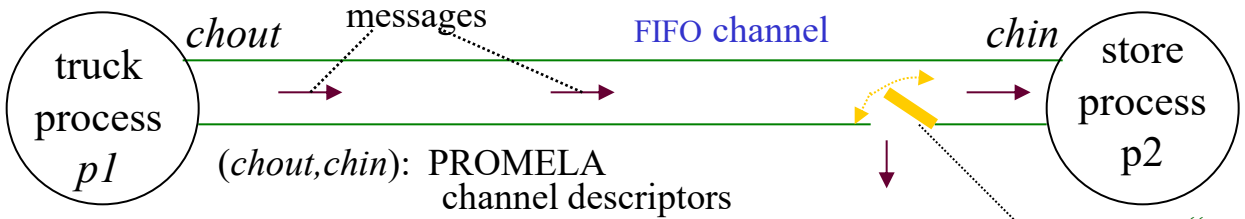


How do we simulate this "accident" in computer model ??

How about a back-up supply truck ??

PROMELA's built-in language constructs

1. "message-channel" with non-lossy FIFO behavior; 2. Guarded statements, with non-deterministic statement selection.



"non-deterministic leak" of messages (to simulate "road accidents")

```
send(x, chout)  →  {receive(x, chin);
                    pass on  x  to process p2;};
send(x, chout)  →  receive(x, chin);
spont(x)        →  send(x, chout);
```

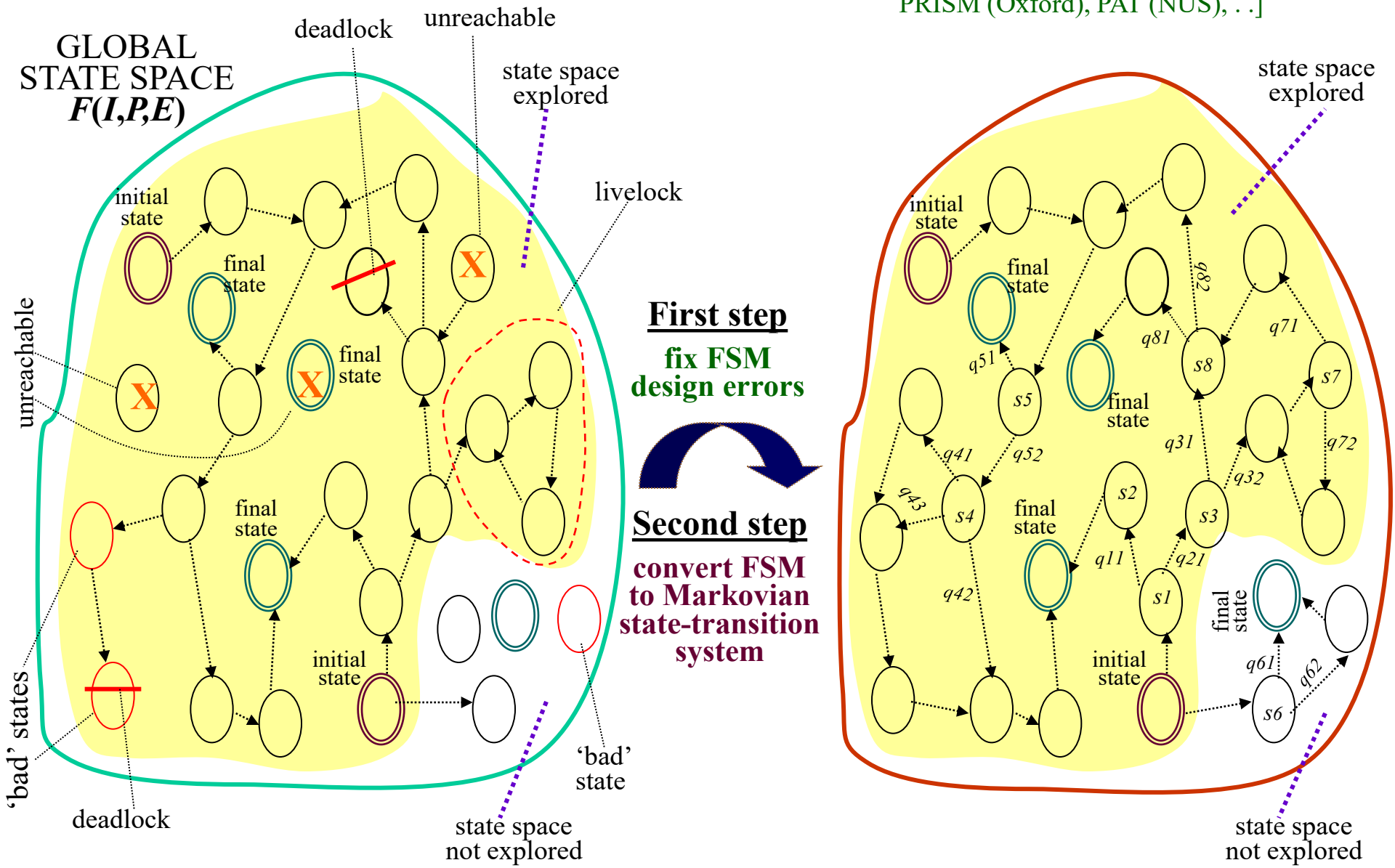
When the guard "send(x,chout)" evaluates to TRUE, the statement blocks to execute is non-deterministically selected by the PROMELA's run-time system SPIN

spontaneous trigger to send message *x*

standard PROMELA based automated algorithm analysis & synthesis

Our extended-PROMELA with probabilistic control of state-transitions

[similar in scope to PRISM (Oxford), PAT (NUS), . .]



coverage = 85.7%
quality of analysis = 89%

state traversal

$q..$: state-transition probability

Case-study of terrain surveillance system by multi-drones with on-board cameras

Accuracy of object detection depends on [camera angle θ, distance *r*] relative to the reference point

Represented as ***confusion probability*** Q_K that depicts camera's detection capability over a set of K objects $(K \geq 2)$

[**confusion probability** Q_K varies with (r, θ)]

Algorithms to analyze UAV camera data for data-fusion

[distributed consensus by voting]

Model-based simulation:
UAV dynamics, resource needs, . .
[battery energy, network bandwidth]

RoS / Gazebo based simulator software

Incentive-based models of sensor software attacks / faults

[use of Economic Theory principles]]

Event occurrences posted by software modules

Discrete-event simulation of composite system behavior under various action plans and control decisions
[Model-checker software tools (e.g., PRISM) and/or operational analysis tools]

UAV camera control [resolution, orientation, . .]

UAV position control

UAV deployment control [add new UAVs, remove current UAVs, reposition UAVs, . .]

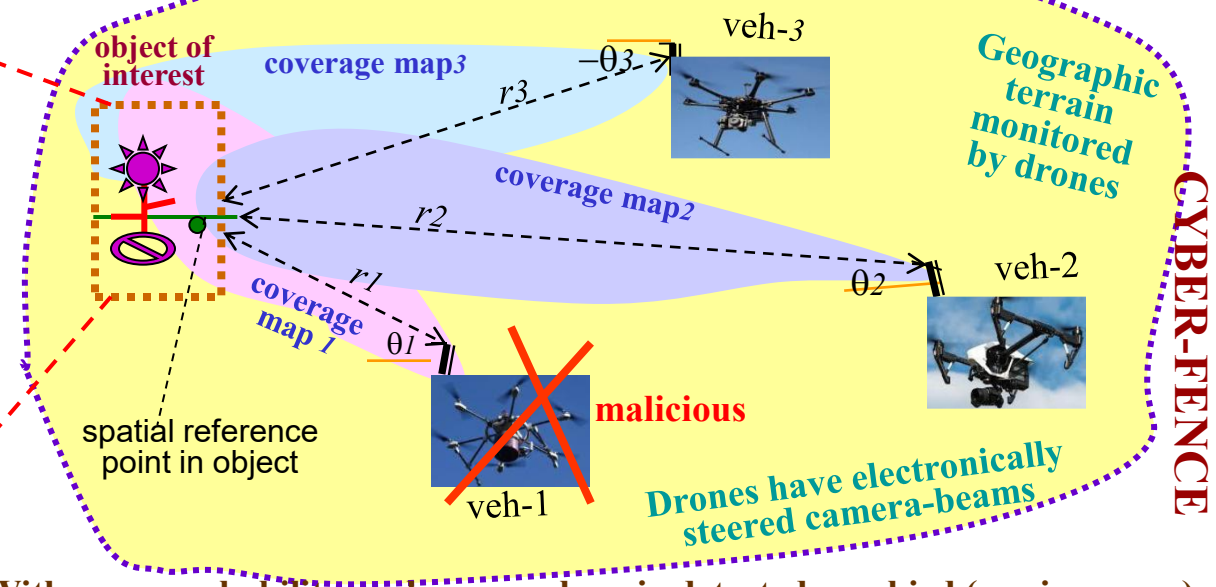
Software-cybernetics interface

Sample images of similar-looking objects in the sky:
plane, bird, baloon, helicopter, . . .

Edge-detection computed by object-recognition algorithm

[distance, view-angle] **vs** coverage quality

fuzzy-data from camera images



With some probability, a plane may be mis-detected as a bird (or vice-versa) by the underlying "data/image comparison" algorithm
➔ **false-positives and false-negatives**

Consensus algorithm as building-block for replicated data-sources

[software prototype runs in CCNY lab (Ack: Support from US Air Force, 2015-18)]

Functional replication of data collecting processes and/or data paths is the key to mask failures, with an appropriate mechanism to manage the replication:

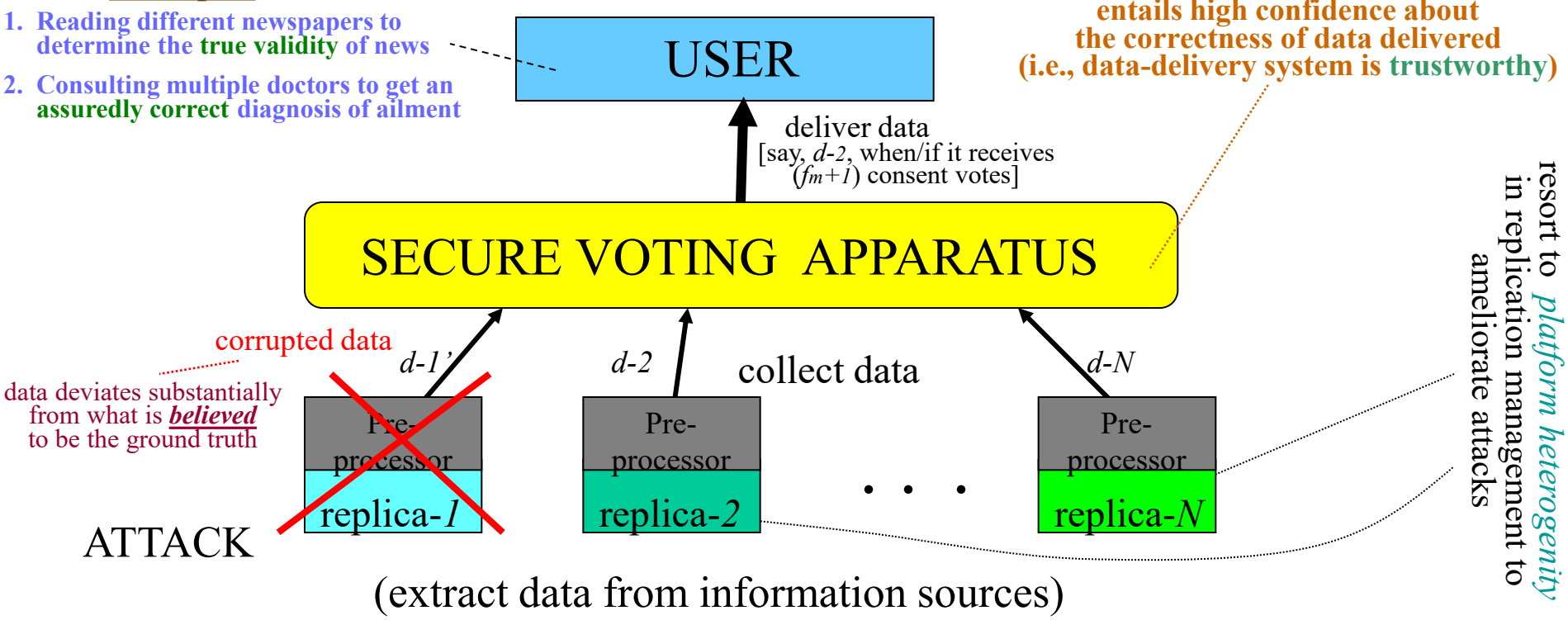
MAJORITY VOTING

[f_m: max. # of faulty replicas; $1 \leq f_m < \lceil N/2 \rceil$

f_m: # of server/device failures that the information system is designed to tolerate
[this is an **assumption** built into the algorithm about the extent of fault occurrences]

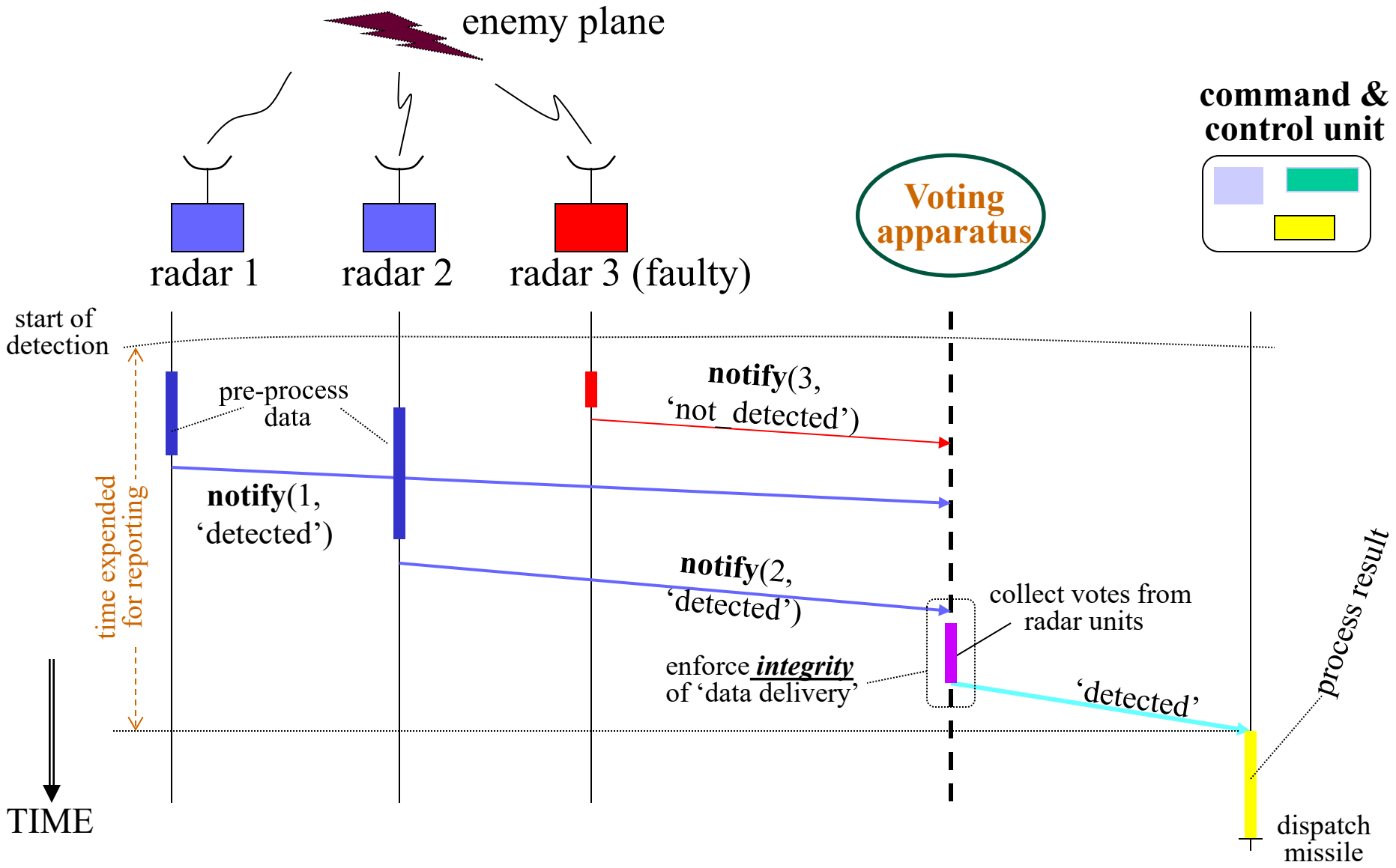
Examples

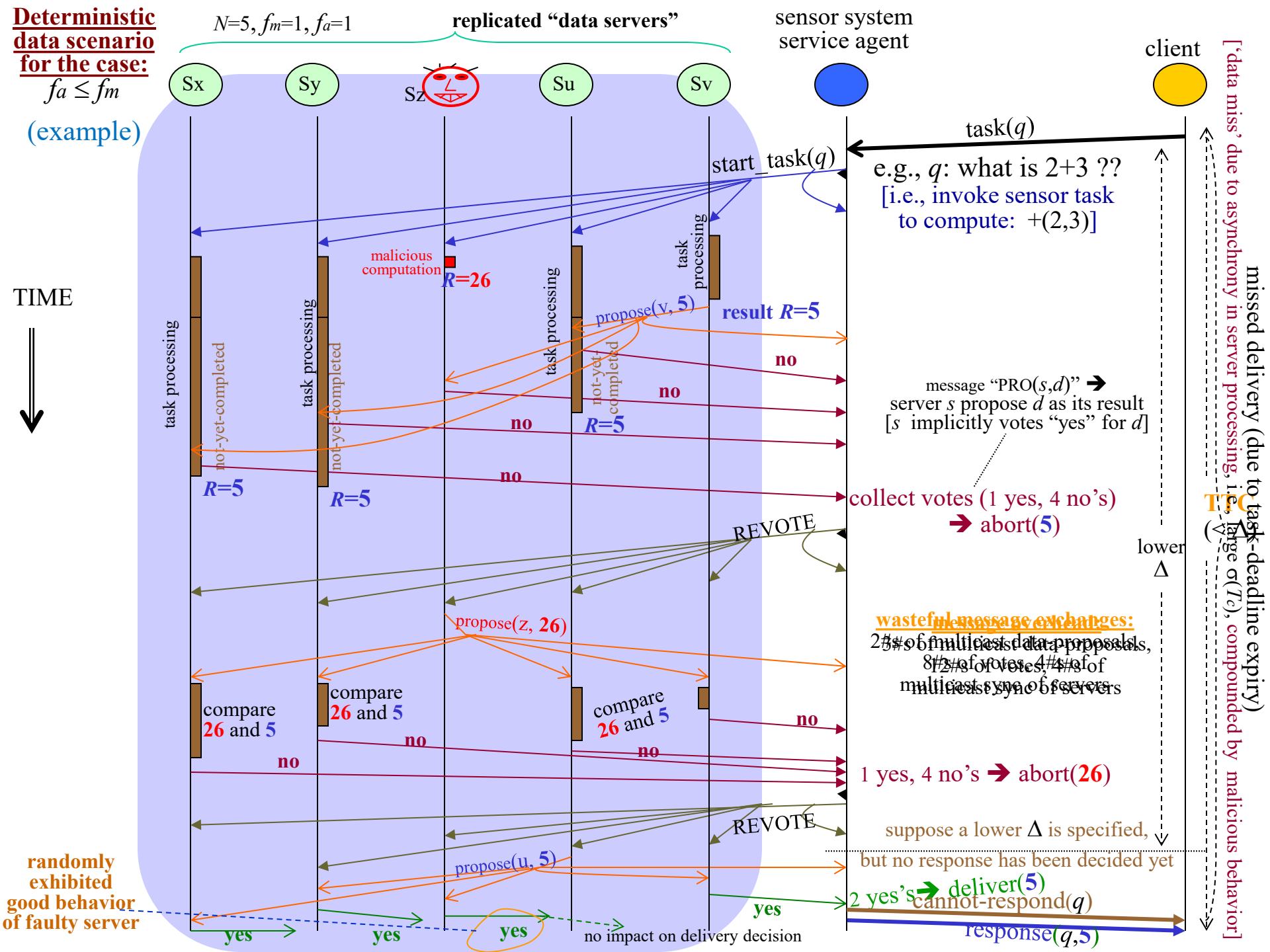
1. Reading different newspapers to determine the true validity of news
2. Consulting multiple doctors to get an assuredly correct diagnosis of ailment

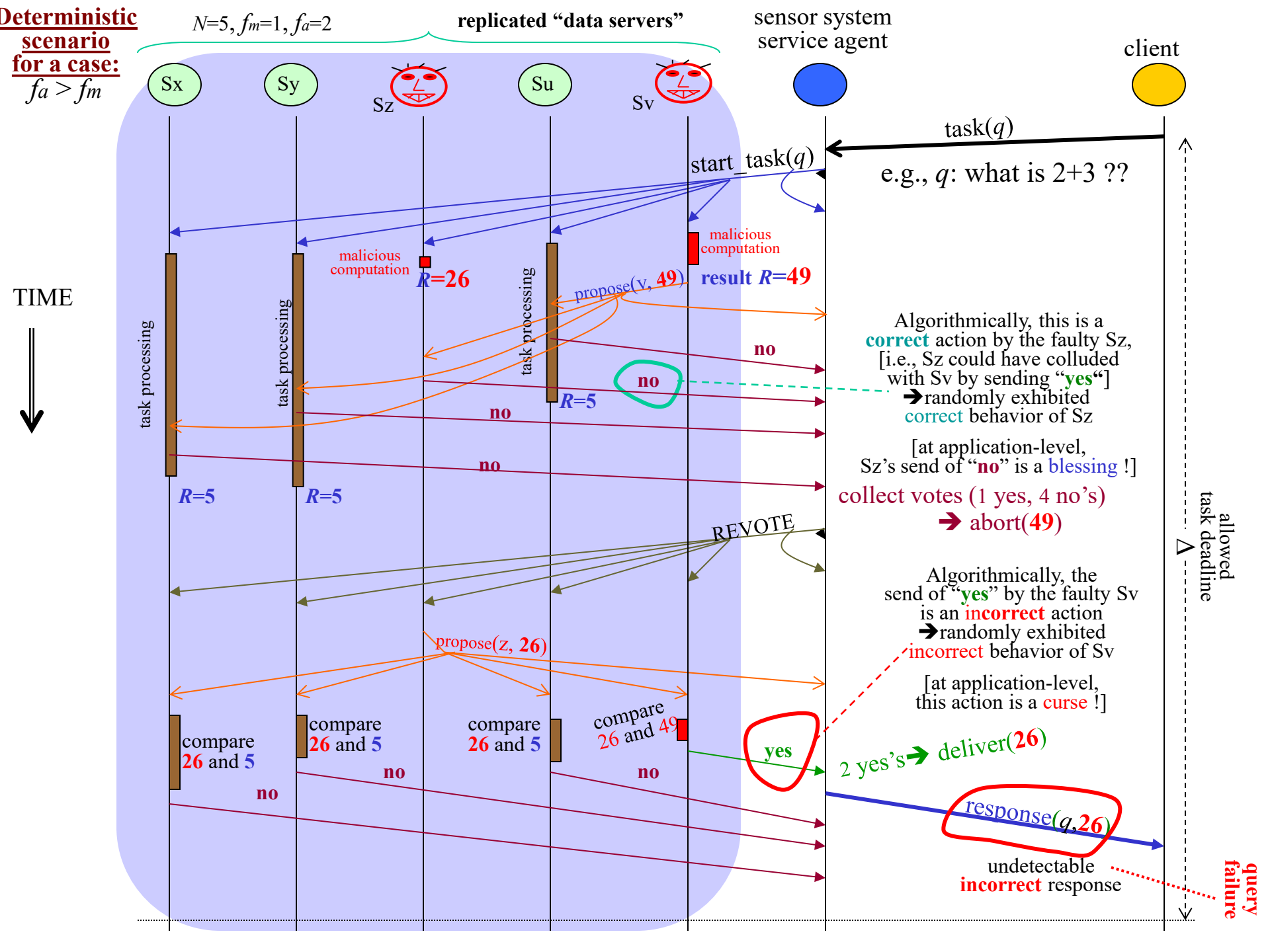


A sample application setting

[to detect enemy planes]



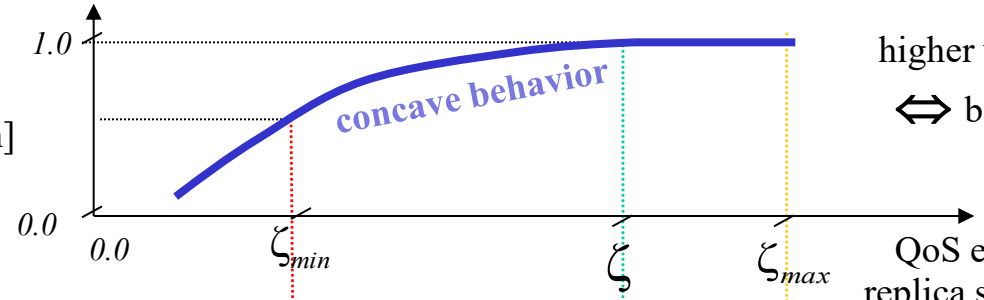

Deterministic data scenario for the case: $fa \leq fm$
(example)
N=5, fm=1, fa=1
replicated "data servers"
sensor system service agent
client
Sx
Sy
Sz
Su
Sv
TIME
task(q)
start_task(q)
e.g., q: what is 2+3 ??
[i.e., invoke sensor task to compute: +(2,3)]
task processing
malicious computation
R=26
result R=5
propose(v, 5)
not-yet-completed
R=5
no
message "PRO(s,d)" ➔ server s propose d as its result
[s implicitly votes "yes" for d]
collect votes (1 yes, 4 no's) ➔ abort(5)
REVOTE
propose(z, 26)
compare 26 and 5
wasteful message exchanges:
message overhead:
multicast sync of servers
1 yes, 4 no's ➔ abort(26)
suppose a lower Δ is specified,
but no response has been decided yet
propose(u, 5)
2 yes's ➔ deliver(5)
cannot-respond(q)
response(q,5)
yes
no impact on delivery decision
randomly exhibited good behavior of faulty server
lower Δ
missed delivery (due to task-deadline expiry)
['data miss' due to asynchrony in server processing, i.e., large σ(Tc), compounded by malicious behavior]


Deterministic scenario for a case: fa > fm
N=5, fm=1, fa=2
replicated "data servers"
sensor system service agent
client
Sx
Sy
Sz
Su
Sv
task(q)
start_task(q)
e.g., q: what is 2+3 ??
TIME
task processing
malicious computation
R=26
malicious computation
result R=49
propose(v, 49)
R=5
no
Algorithmically, this is a correct action by the faulty Sz, [i.e., Sz could have colluded with Sv by sending "yes"] ➔randomly exhibited correct behavior of Sz
[at application-level, Sz's send of "no" is a blessing !]
collect votes (1 yes, 4 no's)
➔ abort(49)
REVOTE
Algorithmically, the send of "yes" by the faulty Sv is an incorrect action ➔randomly exhibited incorrect behavior of Sv
[at application-level, this action is a curse !]
propose(z, 26)
compare 26 and 5
compare 26 and 49
yes
2 yes's➔ deliver(26)
response(q,26)
undetectable incorrect response
query failure
allowed task deadline

Cost *vs* QoS tradeoff in a replicated data-server system

utility value of data service [due to server replication]

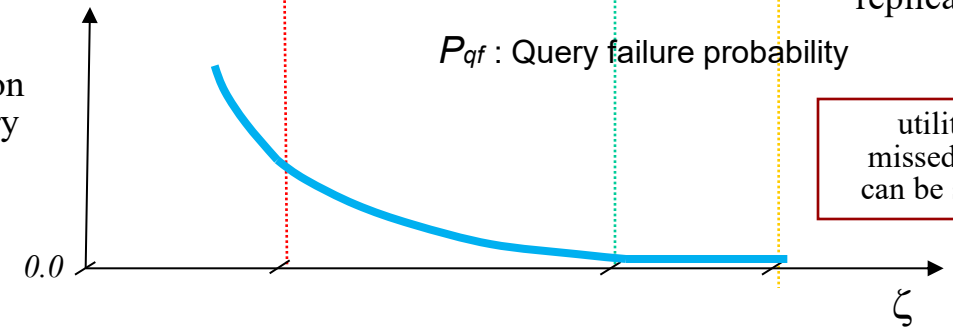
$\boldsymbol{u}\ (\zeta\ ,\ \zeta'\ )$



higher value of ζ ⇔ better QoS

QoS enforced by replica server system (ζ') ..... [$1-P_{qf}$]

penalty measured as user-level dissatisfaction due to bad data delivery



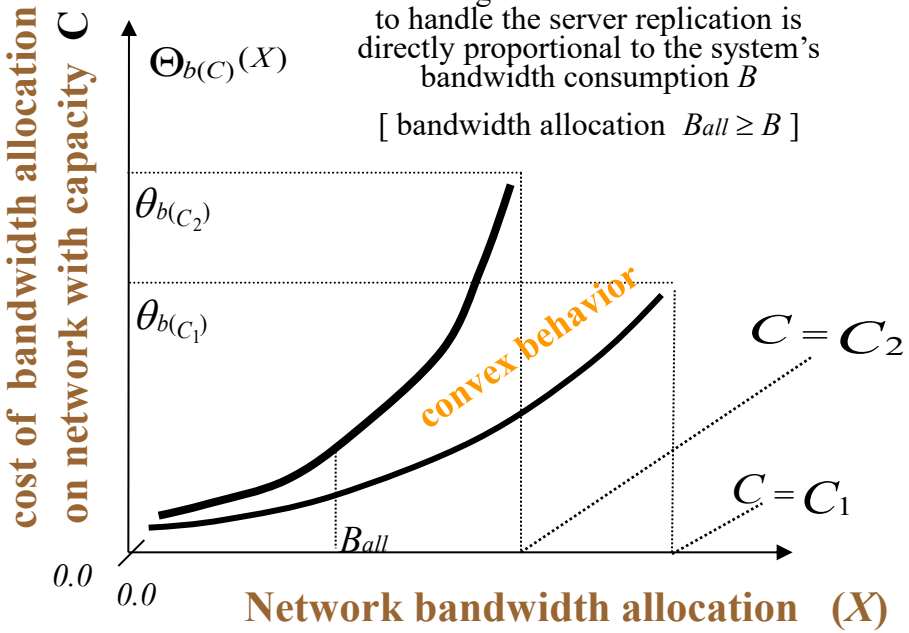
P_{qf} : Query failure probability

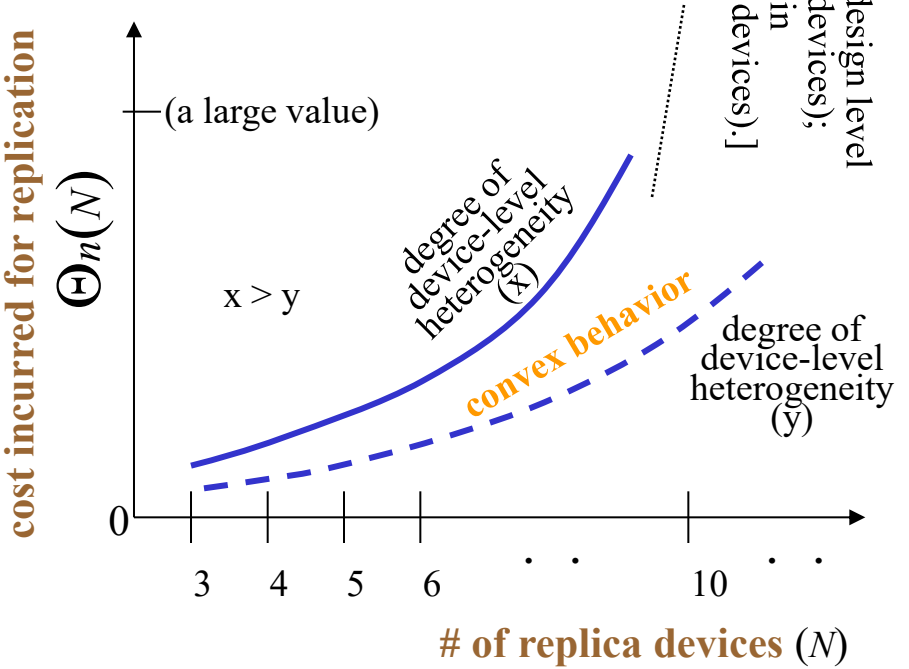
utility function under missed data deliveries P_{qs} can be similarly prescribed

['cost' is determined by the amount of:
1. computational efforts expended at system design level (to replicate devices);
2. deployment/maintenance efforts expended in physical world (to install multiple devices).]

message overhead M incurred to handle the server replication is directly proportional to the system's bandwidth consumption B

[bandwidth allocation $B_{all} \geq B$]



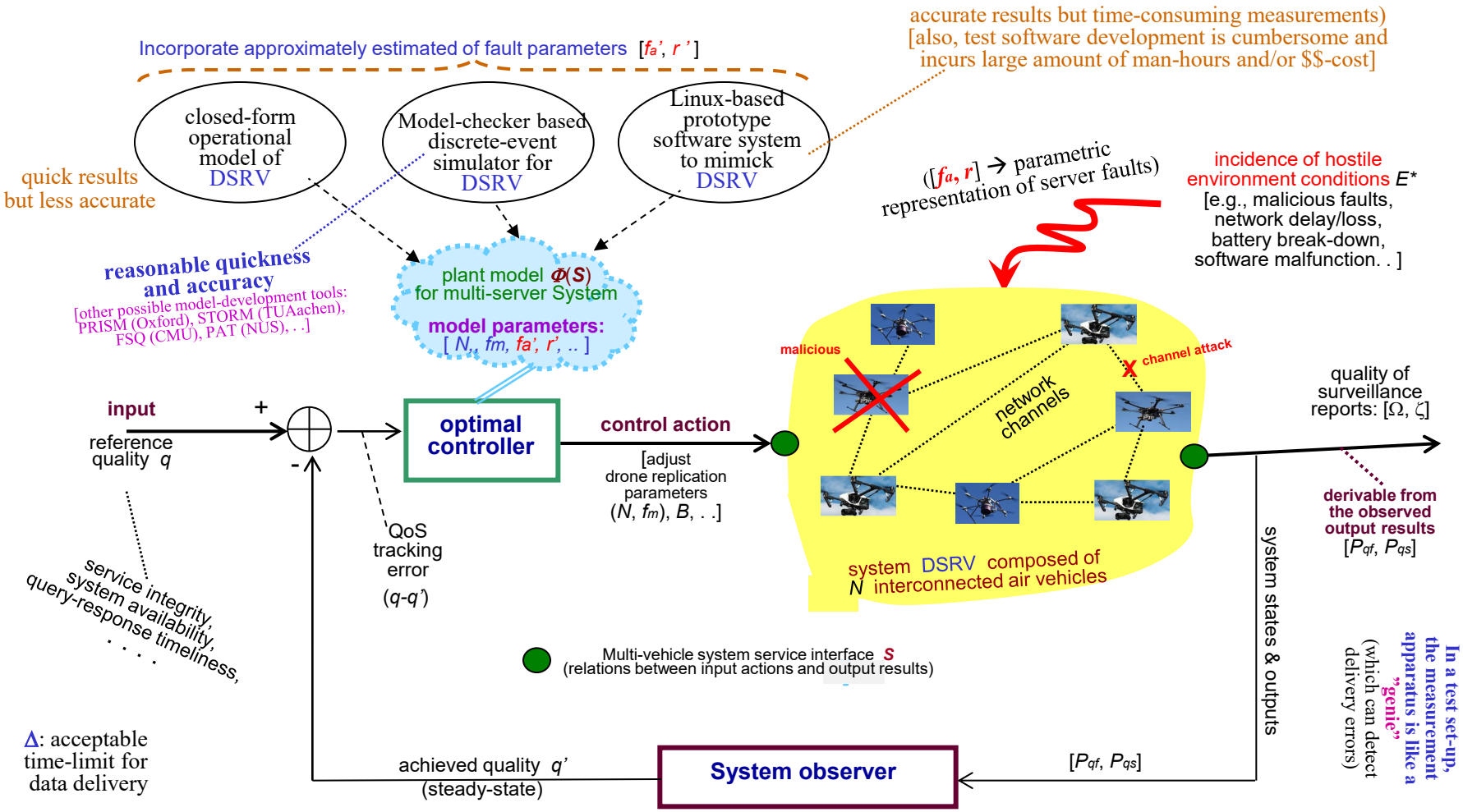


System modeling for: autonomic data-server replication management

Control schema of multi-server query processing system (use AI & ML tools for risk analysis)

Controller solves a constrained optimization problem (to decide on the **control actions** on DSRV system):

Minimize penalty arising from data-service "integrity failures", timeliness violations, and availability degradations --- **subject to** bandwidth and battery-energy consumption and replication costs



Modeling Approach-I:

Employ Operational Models of the Networked System under study

How does an operational model of replica-server system look like ?

system-level ***trust*** metric: Ω
(captures the frequency of service integrity violation)
[derivable from the QoS parameter (*1-Pqf*)]

[*fa*,*r*] are the parameters to be inferred from the observed system output Ω

Query failure probability *Pqf*

$$P_{qf} = \sum_{j=f_m+1}^{f_a} r^j \times (1-r)^{f_a-j} \times 1 - \frac{\binom{(N-j)}{(f_m+1)}}{\binom{N}{(f_m+1)}} \quad (1)$$

$$\text{for } f_m < f_a \leq (N-(fm+1)) \quad (2)$$

$$\sum_{j=f_m+1}^{N-(fm+1)} r^j \times (1-r)^{f_a-j} \times 1 - \frac{\binom{(N-j)}{(f_m+1)}}{\binom{N}{(f_m+1)}} + \quad (3)$$

$$\sum_{i=(N-f_m)}^{f_a} r^i \times (1-r)^{f_a-i} \quad (4)$$

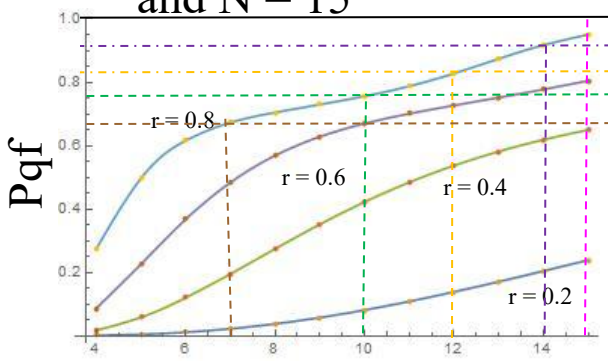
$$\text{for } f_a > (N-(fm+1)). \quad (5)$$

Probability Query Failure vs Number of Faulty servers

N: # of server replicas;
r: failure-severity of a faulty server
(i.e., probability that a faulty server
exhibits an actual failure:
$0.0 < r \leq 1.0$)

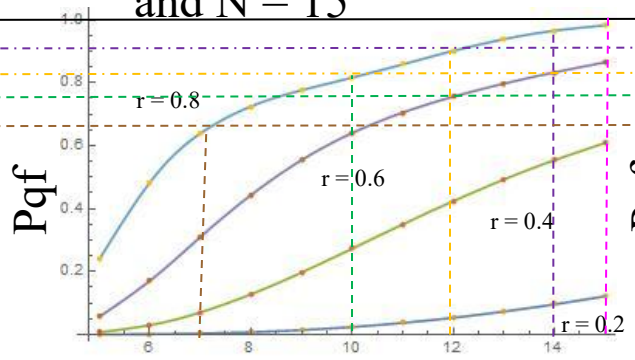
Max. # of faulty servers assumed in the algorithm

Pqf while fm = 3 and N = 15



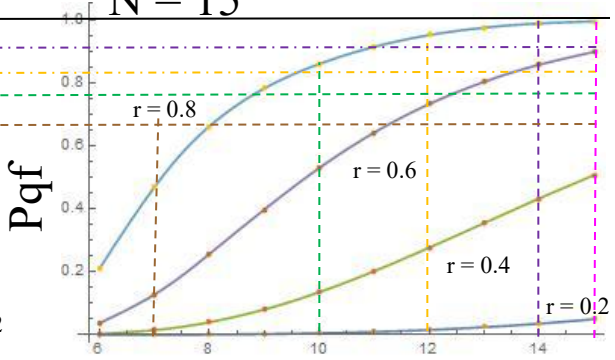
Number of actual Faulty servers (fa)

Pqf while fm = 4 and N = 15



Number of actual Faulty servers (fa)

Pqf while fm = 5 and N = 15



Number of actual Faulty servers (fa)

Sample set of experimental results, as validated by mathematical model*

Actual [N,fm] unknown to test designer

Claimed: [N,fm] = [9,3]

Observed L': 4.0

Model:: threshold: 0.155

[fa,r] known to test designer

[fa=2, r=0.5]		
N	fm	L'
6	2	3.27702
7	2	2.95272
8	2	2.70147
	3	4.13931
9	2	2.50412
	3	3.83729
10	2	2.34707
	3	3.57997
	4	5.09098

Step 1: Explore [N,fm] parameter space and compute L';

Identify L' that is(are) within the threshold of the observed L'

	Actual	Model::	threshold=0.14
	Pqf	Pqf	Diff
[N=9,fm=3,fa'=4,r'=0.8]	0.25336	0.39334	0.13998
[N=8,fm=3,fa'=4,r'=0.8]	0.2615	0.40374	0.14224

Step 2: Compute Pqf for the two cases to disambiguate between them:

Test designer introduces known fault fa',r'] to a test query in the actual system

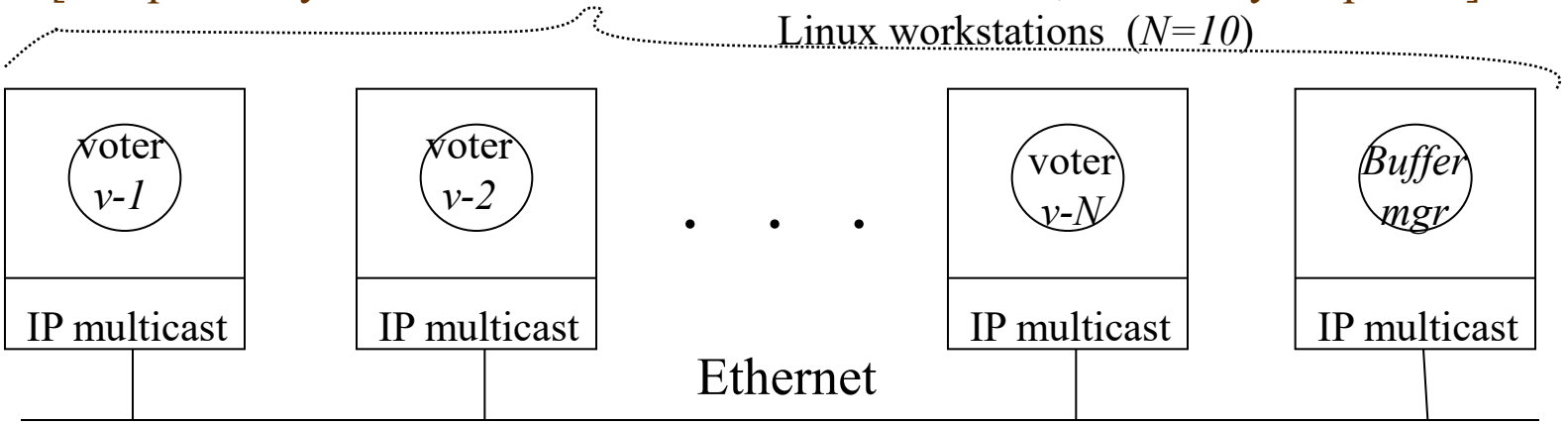
- IEEE INFCOM '26 (Ravi and etal)
- IEEE/IFIPAnnet'19 (Arun, Ravi)

Modeling Approach-II:

Employ Software Prototypes of Networked System under study

Prototype implementation of replica-server software system at CUNY

[completed by a team of 4 Ph.D. and 3 M.S. students, over a 2-year period]



Scope of prototyping efforts

** Functionality checks and protocol validation by 'stress testing'

Subject the protocol to rigorous failure scenarios to ascertain its 'correct data delivery' capability and identify any design errors therein

** Performability checks on protocol

Determine the maximum achievable performance by testing the protocol under 'normal case' behaviors in the environment (i.e., message loss is very low, $\sigma(Tc)$ is small, and $fm \ll N/2$)

^^ Experiments with random injection of faulty voter behaviors, control of voter asynchrony by random spread out of the 'data ready' time of servers

^^ IP multicast transport, for the purpose of performability studies

Sample experimental results on replica system performance

N=10;

# of consent votes needed = *6;*

$\sigma(T_c)$=*50 mec;* $\mu(T_c)$=*50 msec;*

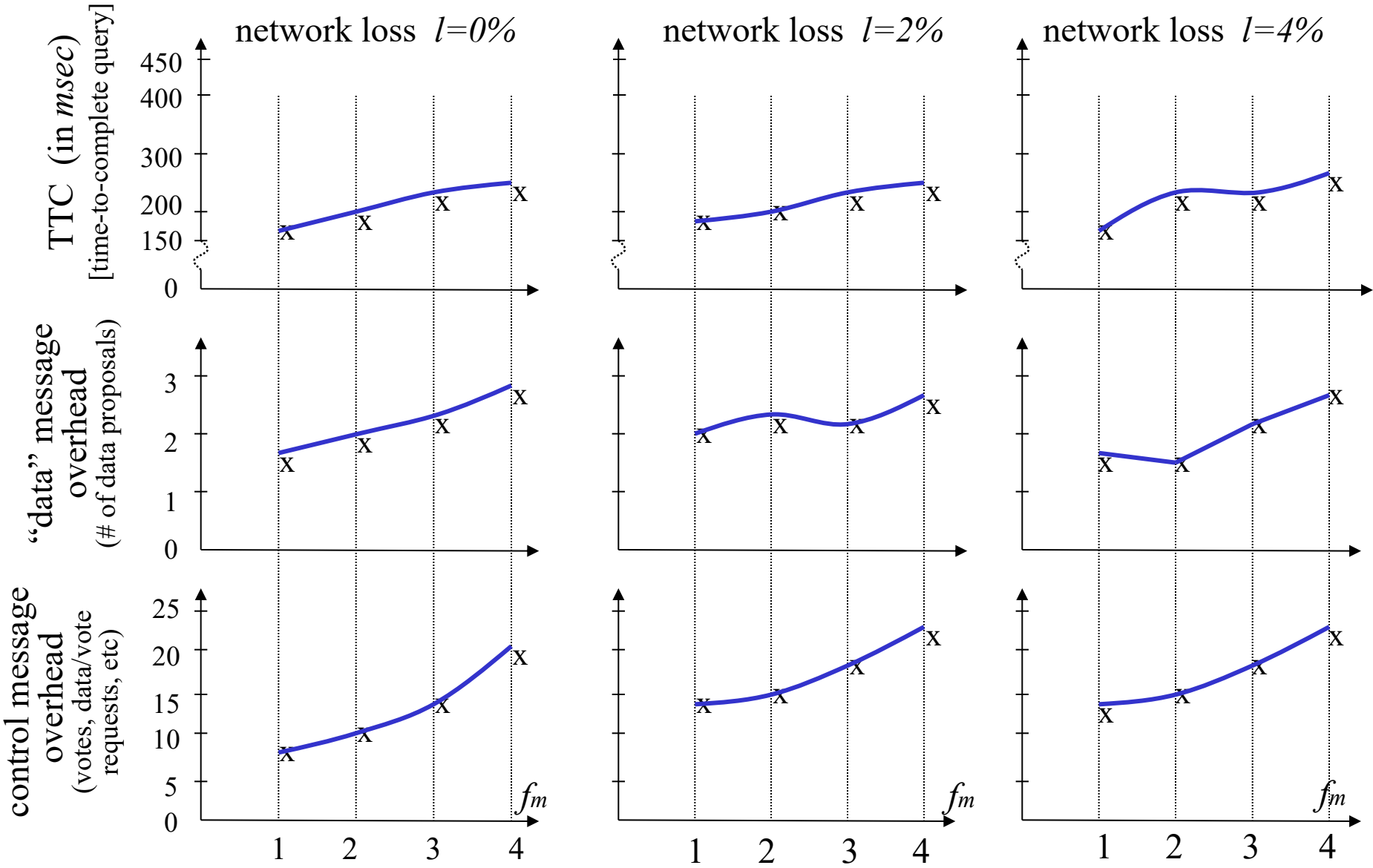
data size = 30 kbytes

control message size = 50 bytes

f_m : algorithmic assumption on # of faulty servers

f_a : actual # of faulty servers

$f_a = f_m$ in our experiments



Modeling Approach-III:

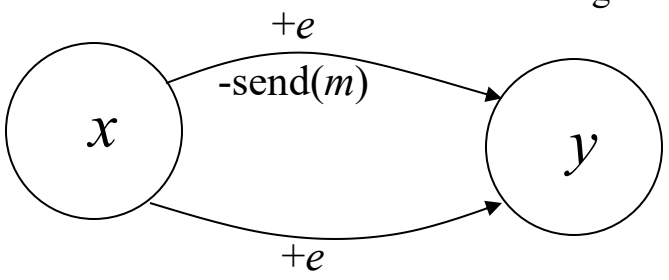
Employ Probabilistic model-checkers of Networked System under study

[illustration of model-checking with extended-PROMELA (with software-overlays)]

FAULT: *"SEND-OMISSION of process"*

FSM of 'network_sender'

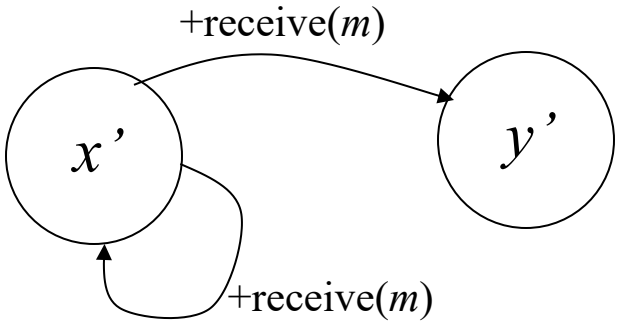
e: trigger for data generation



state = *x* && event(*e*) → {send(*m*); state := *y*;};
state = x && event(*e*) → state := *y*;

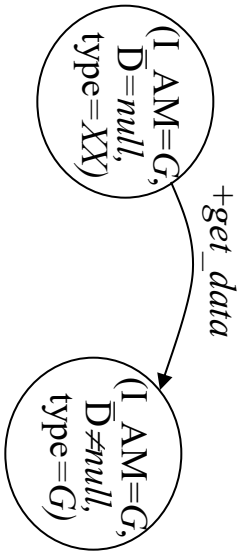
FAULT: *RECEIVE-OMISSION of process*

FSM of 'network_receiver'

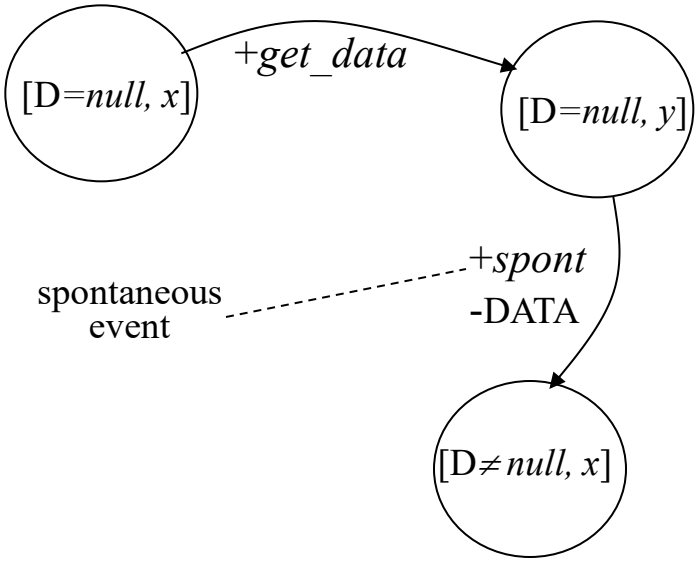


state = *x'* && receive(*m*) → state := *y'*;
state = x' && receive(*m*) → ;

FAULT: "data corruption"



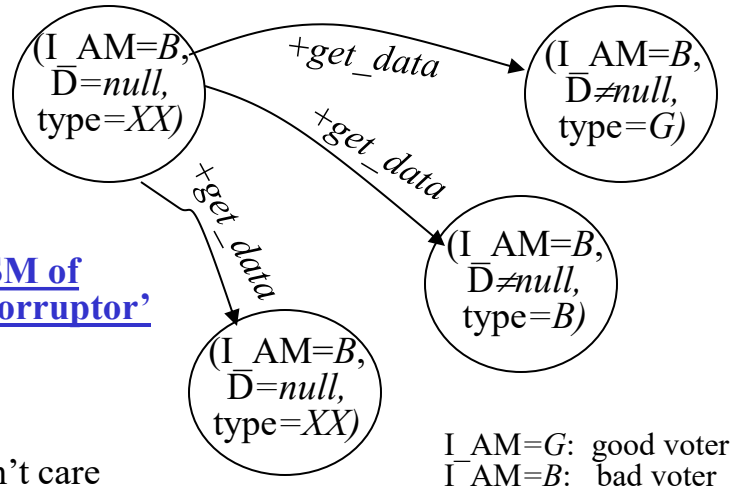
FAULT: "asynchronous data generation"



FSM of 'asynchronous data generator'

[D = *null*, state=*x*] && event(*get_data*) → {state := *y*;};
[D = *null*, state=*y*] && event(*spont*) → {D ≠ *null*; state := *x*;};

FSM of 'data corruptor'



xx: don't care

I_AM=*G*: good voter
I_AM=*B*: bad voter

(I_AM=*xx*, D = *null*) && event(*get_data*) → {D ≠ *null*; type := *G*;};
(I_AM=*B*, D = *null*) && event(*get_data*) → {D ≠ *null*; type := *B*;};
(I_AM=*B*, D = *null*) && event(*get_data*) → ;

Capturing faulty behaviors in PROMELA-model of replicated servers

N: # of deployed servers; f_m: max. # of servers ***assumed*** as faulty; f_a: # of ***actual*** faulty servers;
r: faulty-severity of a faulty server; K: # of deployable servers (say, # of cloud-leased servers).

$$(f_m < f_a \le N; \quad 0.0 < r \le 1.0; \quad N \le K; \quad 1 \le f_m < \lceil N/2 \rceil)$$

Initial set up of hostile environment

```
for (i := 1;  i ≤ fa; i++)
      server_type[i] := BAD;  /* mark server as faulty */
      server_severity[i] := r;
if (fa < N)
    for (i := fa+1;  i ≤ N;  i++)
        server_type[i] := GOOD;  /* mark server as non-faulty */
```

Issue: How do we set up the probabilistic behavior of a faulty server Z (say, $0.0 < r < 1.0$) ??
[i.e., Z proposes a corrupted data with probability r, and proposes a non-corrupted data with probability $(1\text{-}r)$]

Basic set up for each server $j \mid _{j=1,2,..,N}$ in standard-PROMELA

'data trigger' conditions for server j are not shown in guard expressions

Non-deterministic guard (expressed in PROMELA)

```
(server_type[j] == GOOD):
        {generate data x;
         x.label := NC;} /* mark data as 'not corrupted' */
(server_type[j] == BAD):
        {generate data x;
         x.label := C;} /* mark data as 'corrupted' */;
(server_type[j] == BAD):
        {generate data x;
         x.label := NC;}
```

➔ $r = 0.5$ is the only possible assignment

Infusion of probabilistic behaviors in replica server systems

"meta-label" set up for each server j | $j=1,2,..,N$

(for **observational** purposes)

(server_type[j] == **GOOD**) $\wedge$ (dtrig[j] == TRUE):
{generate data x;
x.label := NC; /* mark data as 'not corrupted' */
dtrig[j] := FALSE;}

guards, expressed in PROMELA

(server_type[j] == **BAD**) $\wedge$ (dtrig[j] == TRUE):
{generate random # $R \in (0,1]$;
($R \leq r$):
{generate data x;
x.label := C; /* mark data as 'corrupted' */
dtrig[j] := FALSE;}

sub-guards

dtrig[j]: external trigger event for server j to generate data [e.g., end of server computation, malicious action, etc]

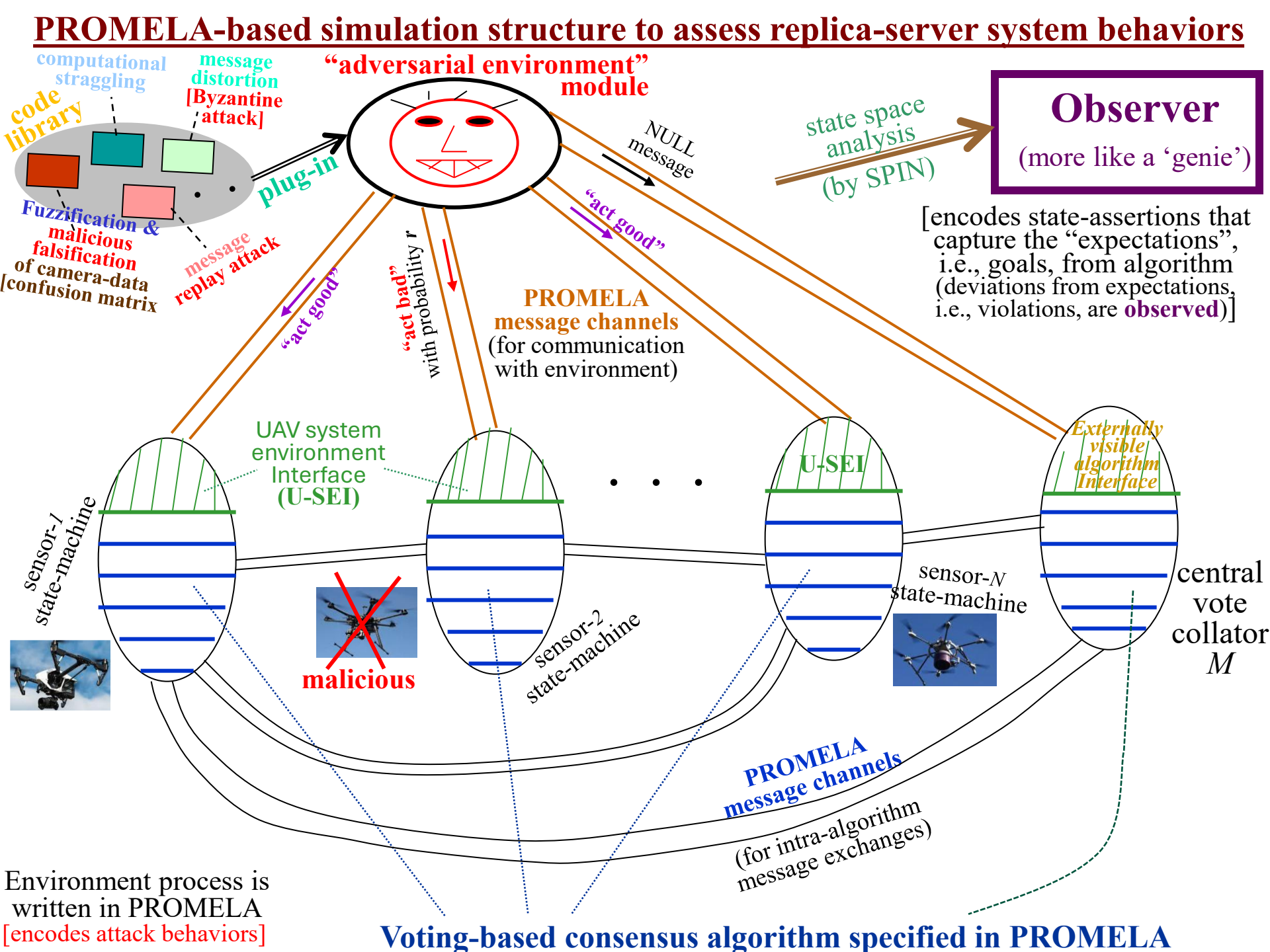
(server_type[j] == **BAD**) $\wedge$ (dtrig[j] == TRUE):
{generate random # $R \in (0,1]$;
($r < R \leq 1.0$):
{generate data x;
x.label := NC;
dtrig[j] := FALSE;}

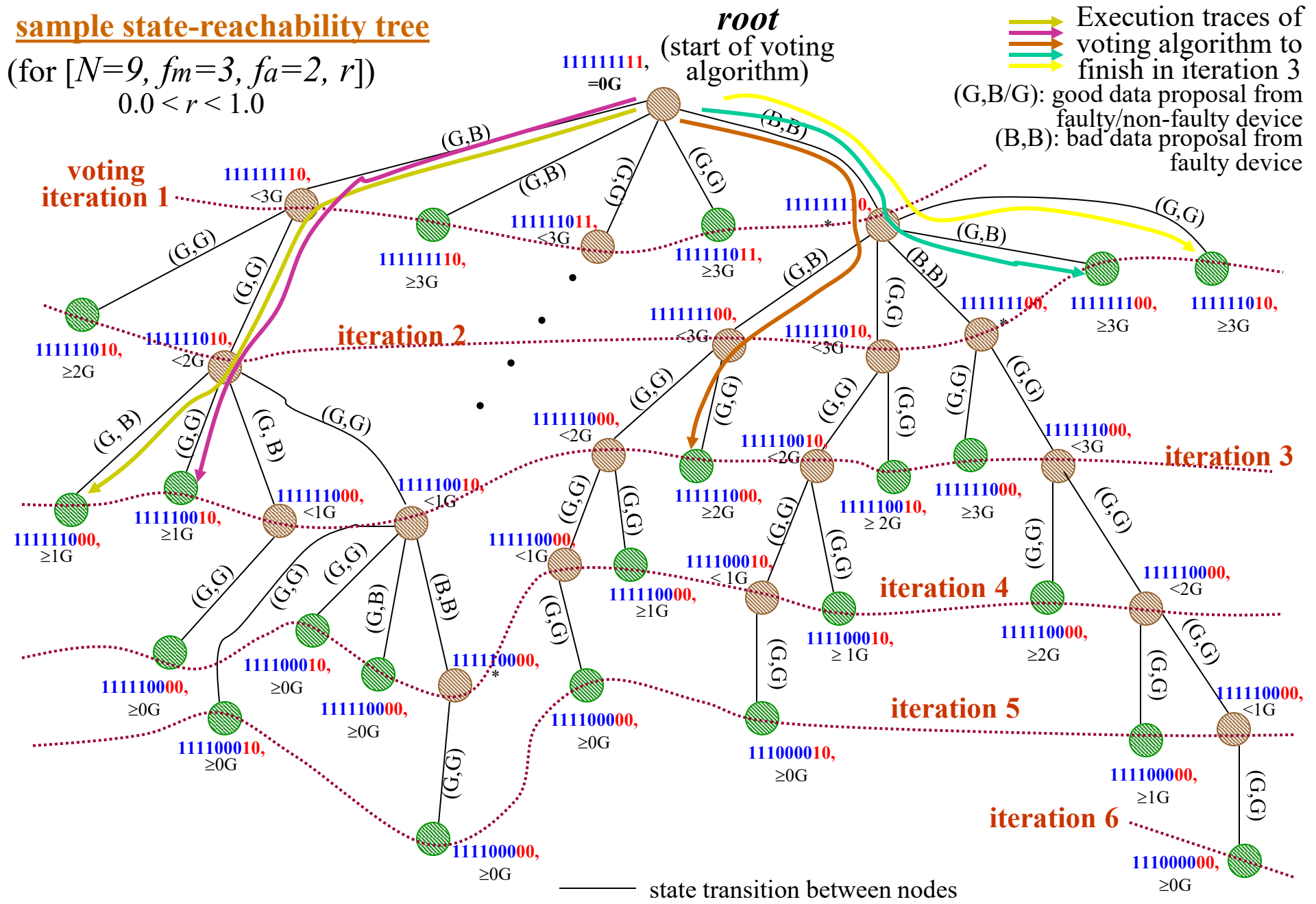
Transactional query failure probability:

$$P_{qf} = \frac{\text{\# of deliveries with NC label}}{\text{Total \# of deliveries (i.e., both NC and C)}}$$

counted by SPIN from state-traces

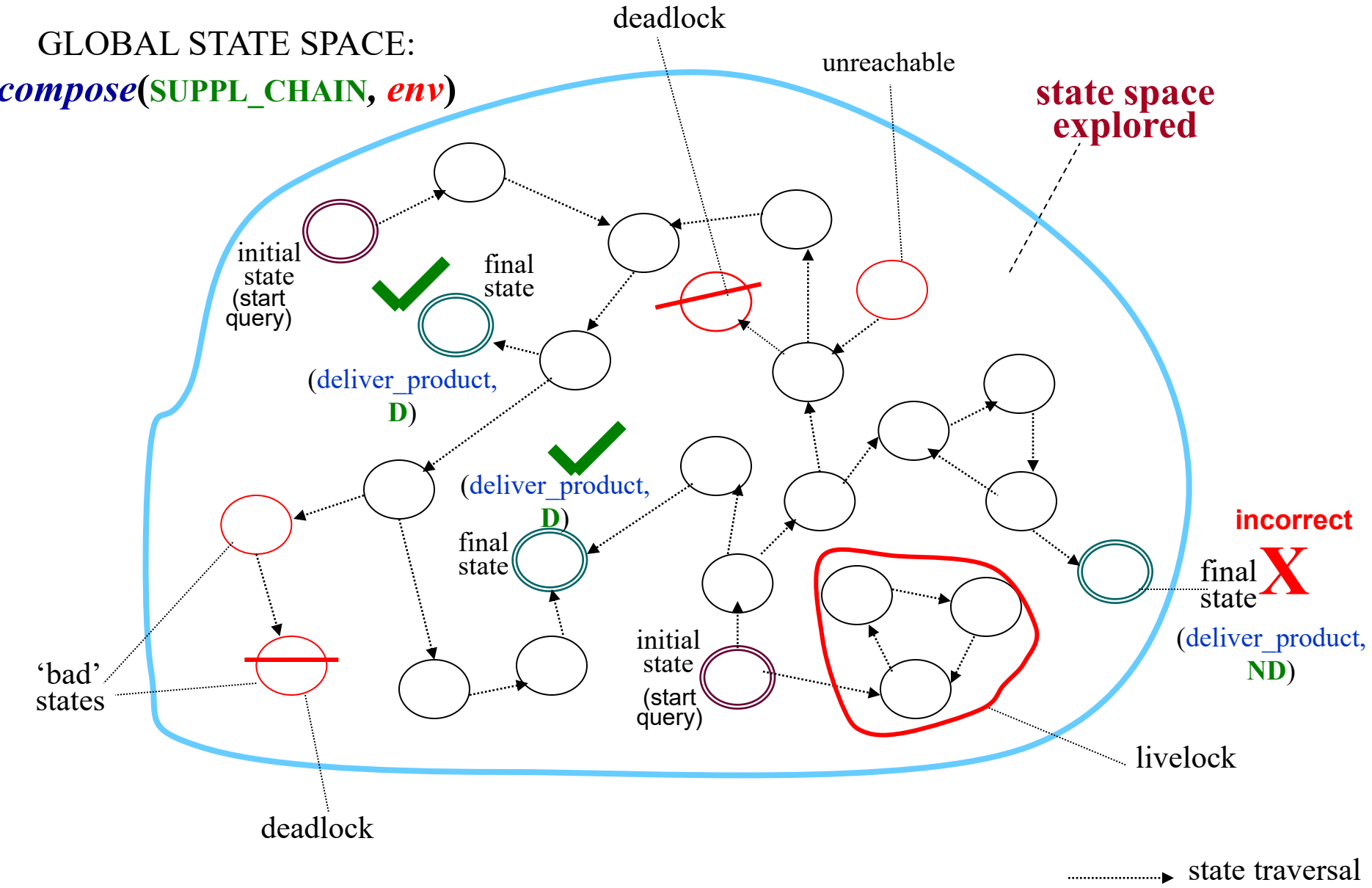
PROMELA-based simulation structure to assess replica-server system behaviors

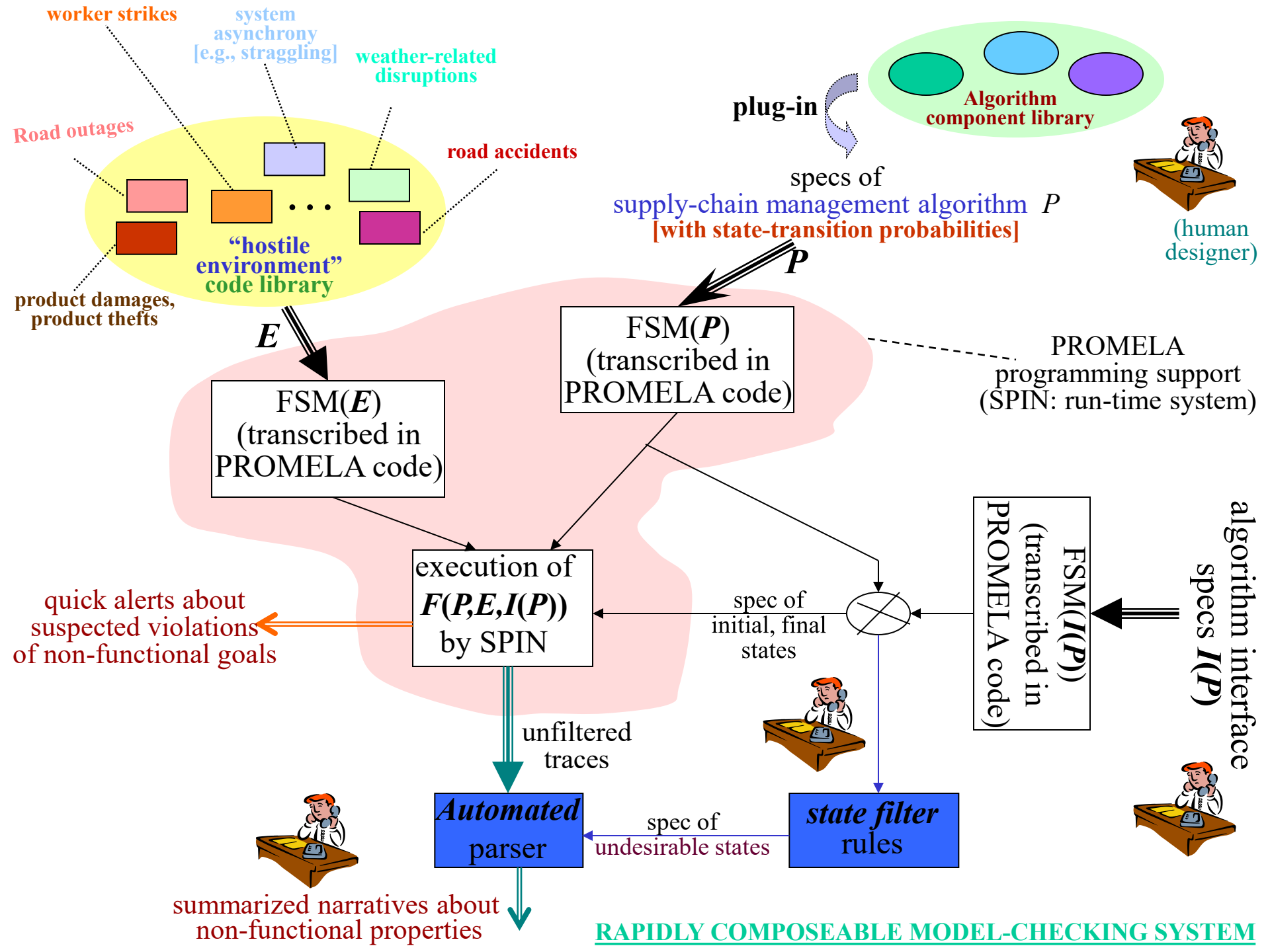



sample state-reachability tree
(for [N=9, fm=3, fa=2, r])
0.0 < r < 1.0
root
(start of voting algorithm)
111111111, =0G
Execution traces of voting algorithm to finish in iteration 3
(G,B/G): good data proposal from faulty/non-faulty device
(B,B): bad data proposal from faulty device
voting iteration 1
iteration 2
iteration 3
iteration 4
iteration 5
iteration 6
state transition between nodes
Successful iteration (i.e., data delivery completes)
Aborted iteration (due to insufficient votes)
Node is marked with tuple: <bit-map of devices eligible to propose, # of additional devices with good data G>

Quick ideas on probabilistic model-checking for resilient supply-chain networks

Use of meta-labels D and ND to track good and bad product deliveries during supply-chain simulated execution




worker strikes
system asynchrony [e.g., straggling]
weather-related disruptions
Road outages
road accidents
"hostile environment" code library
product damages, product thefts
plug-in
Algorithm component library
specs of supply-chain management algorithm P
[with state-transition probabilities]
(human designer)
P
E
FSM(P) (transcribed in PROMELA code)
FSM(E) (transcribed in PROMELA code)
PROMELA programming support (SPIN: run-time system)
execution of F(P,E,I(P)) by SPIN
quick alerts about suspected violations of non-functional goals
spec of initial, final states
FSM(I(P)) (transcribed in PROMELA code)
algorithm interface specs I(P)
unfiltered traces
Automated parser
spec of undesirable states
state filter rules
summarized narratives about non-functional properties
RAPIDLY COMPOSEABLE MODEL-CHECKING SYSTEM

Summary of our works on probabilistic model-checking

- PROMELA software overlays to assign state-transition probabilities and SPIN trace-analysis
- Rapid composition of algorithm functions into PROMELA processes
- Exploration of other model checkers: PRISM (Oxford), PAT (NUS), STORM (Tech Univ Aachen), . .
- Extraction of probabilistic measures of system adaptation processes
- Machine-intelligence and Markov decision tools for system analysis in the face of uncertain events
- Autonomic methods for system-level reconfigurations (such as switching between pessimistic and optimistic algorithms)