Dependability of Adaptable and Evolvable Distributed Systems

EU Project Dossier
Summer School
Cyprus, Limassol

September 2016

Carlo Ghezzi
DEIB-Politecnico di Milano

Outline

- Of software and change
- Evolution, adaptation, self-adaptation
- How can they supported dependably?
- How can dynamic evolution be supported for continuously running systems?

Software and change

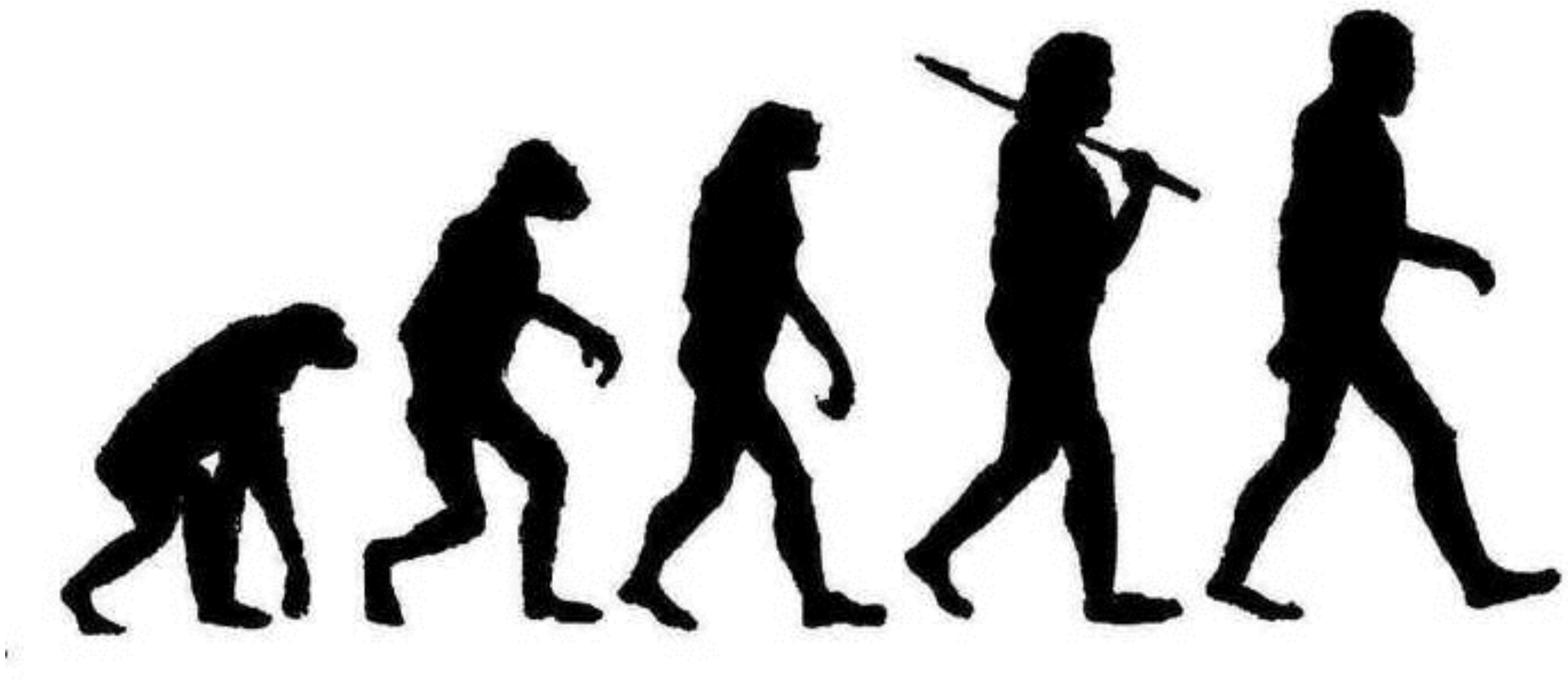
Facts

- Software undergoes continuous changes
- Unrivalled by any other technology
- Can be a problem
- **Can be an opportunity**

Evolution: positive view of change

Embeds the notions of

- improvement
- adaptation



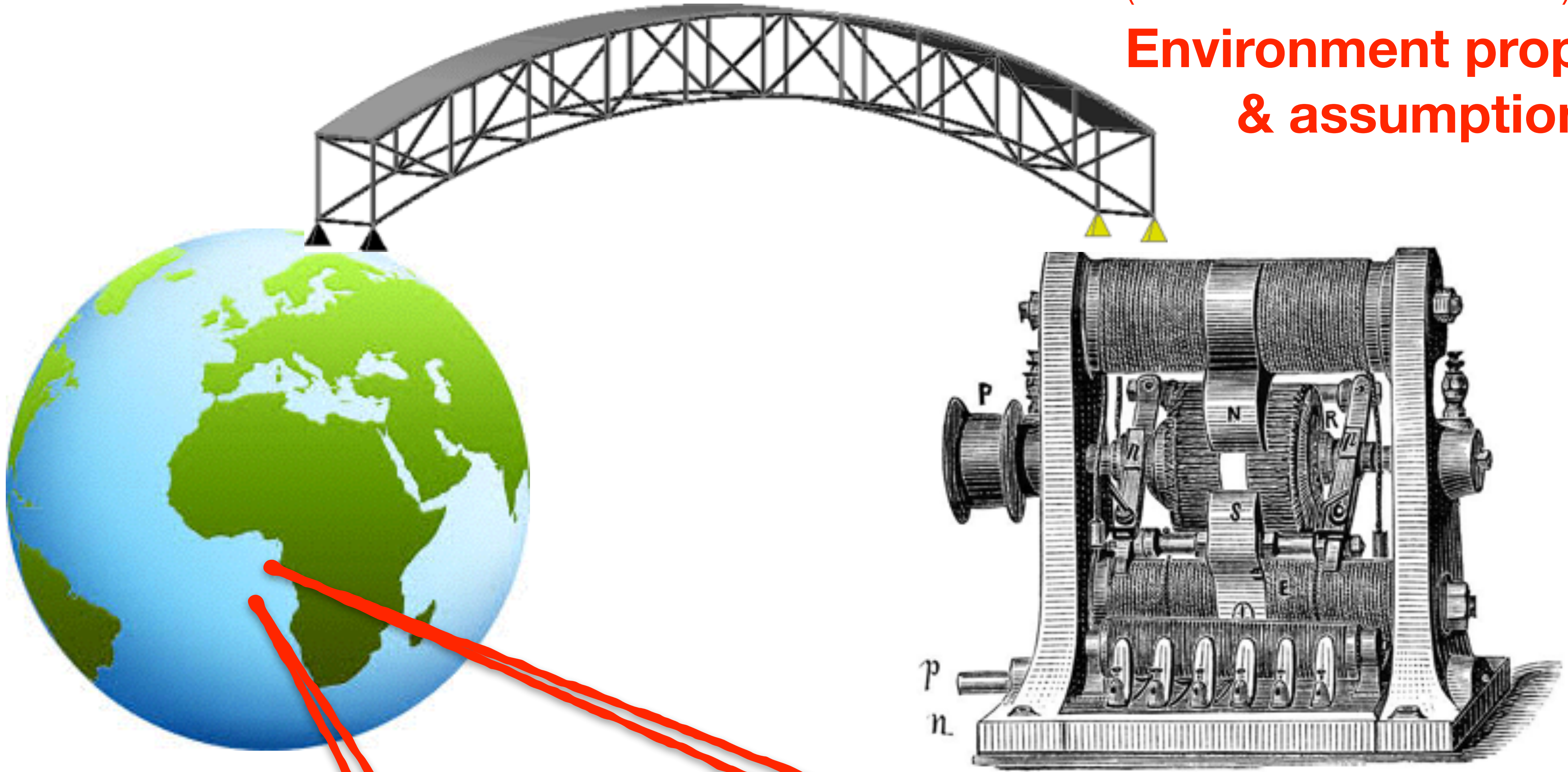
Formal methods can be integrated into software engineering practice to achieve dependability and effectively and efficiently turn change into evolution

Why change?

The world and the machine

… they bridge the gap between requirements and specifications (M. Jackson & P. Zave)

Environment properties & assumptions



We build (abstract) machines to achieve certain real-world goals, satisfy certain requirements

P. Zave, M. Jackson. Four dark corners of requirements engineering. *ACM Trans. Softw. Eng. Methodol.* 6, 1 (January 1997)

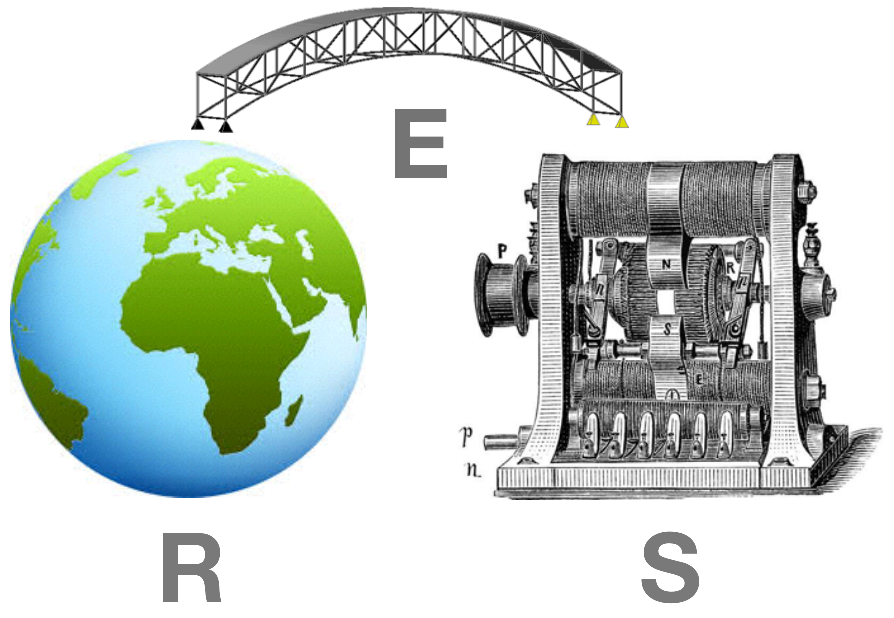
Software engineer's responsibilities

- Develop a specification **S** for the machine (and an implementation) such, assuming that the environment behaves according to **E**, we can assure satisfaction of **R**

- Formally,

$$\mathbf{E} \;\&\; \mathbf{S} \models \mathbf{R}$$

Dependability argument

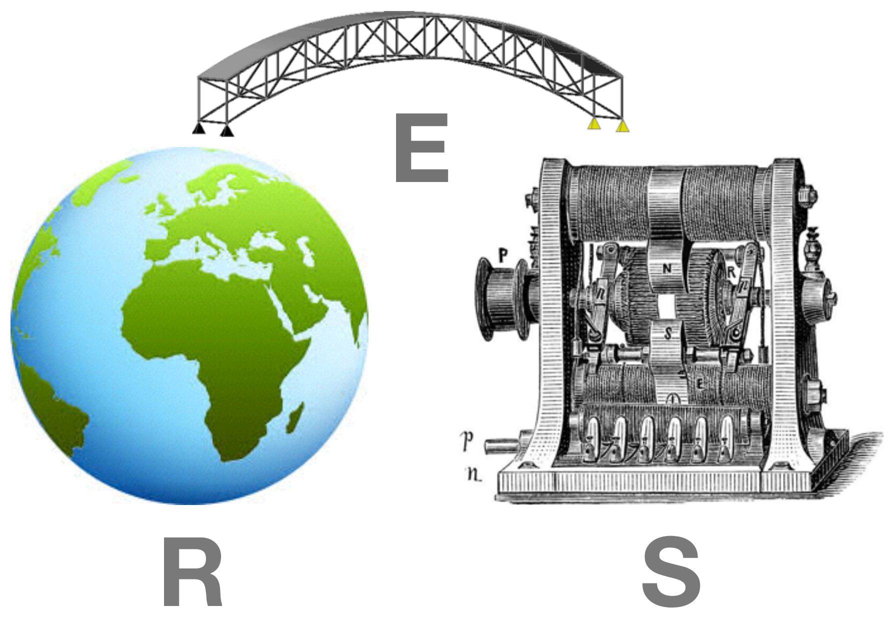


Change source: Getting the machine right

- It is an evolutionary process
- Software design is an exploratory activity
- Software evolves from incomplete to a progressively complete and stable solution
- An then the solution becomes unstable to support further evolution

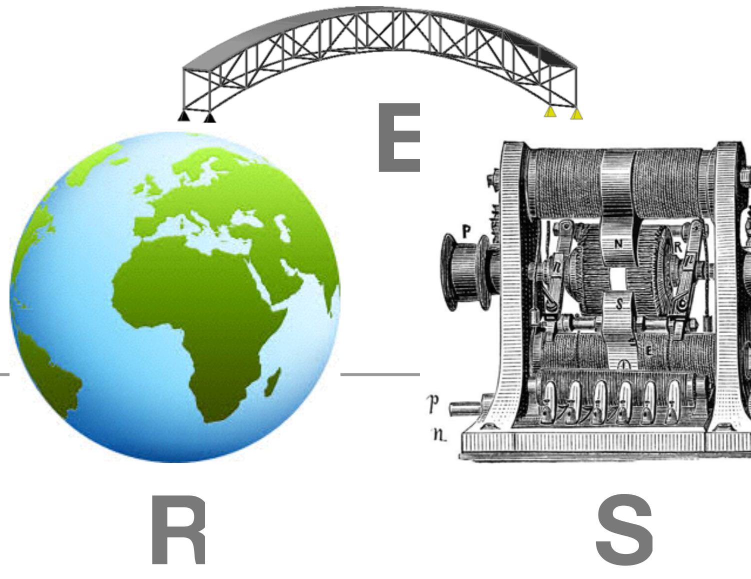
Change source: Requirements

- Requirements are highly volatile
  - Hard to get
  - Change rapidly
  - Satisfaction of certain requirements generate new requirements



E & S ⊨ R

Change source: The environment



- Getting environment properties and assumptions right is hard
  - **Properties**
    - Domain laws (e.g., physics)
    - **R**: move a body from A to B
    - **D**: suitable force causes motion A->B
      - **S**: send suitable force command to actuator
  - **Assumptions**
    - Uncertain/incomplete/changeable knowledge
    - **R**: guarantee avg. response time to users
    - **D**: avg. traffic X transactions/sec.

More on properties vs. assumptions

- Property
  - it holds regardless of any software-to-be; e.g. physics' laws

    avgTrainAcceleration (t1, t2) > 0 implies trainSpeed (t2) > trainSpeed (t1)

- Assumption
  - may expect a violation

    "temperature is in the range -40..+40 Celsius"

    "device generates a measure every 2 ms."

    "humans behave as instructed by the machine"

Change source: The environment

- Often wrong properties/assumptions are hypothesized
- Often assumptions made at design time are uncertain
- Often assumptions change

$$\mathbf{E} \,\&\, \mathbf{S} \models \mathbf{R}$$

The (in)famous Airbus accident (Sept. 1993)

- **Requirement**: ReverseThrust —> TouchedDown
- **Machine Spec**: ActuateRevThrust —> WheelPulsesOn
- **Assumptions**:

WheelPulsesOn <—> WheelsTurning

ActuateRevThrust <—> ReverseThrust

Sensor/actuator correctness assumption

WheelsTurning <—> TouchedDown

OK?

The notion of failure

- Failure = broken dependability argument
- Functional or nonfunctional failure
  - not necessarily a "catastrophic event"
  - includes violation of **quality of service**, which may lead to financial losses, penalties, or damage to reputation
- ***Experienced or predicted failures drive evolution***

A relevant case: multi-owner systems

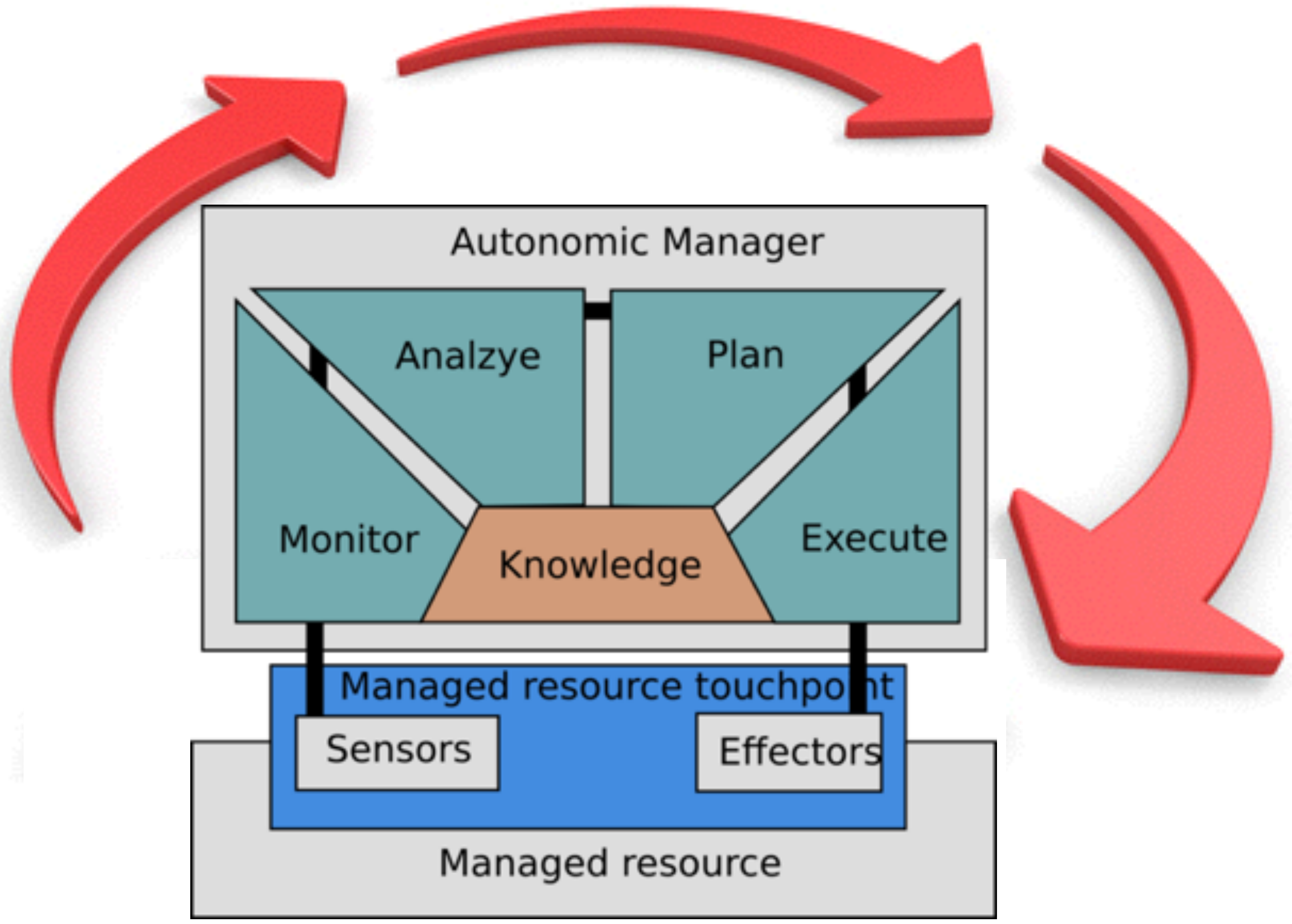
- Rely on third-party components to provide their own service, which make environment volatile
  - Platform as a Service (cloud)
  - Software as a Service
  - Reinvigorating Leslie Lamport's statement
    - *a distributed system is a system where I can't get my work done because a computer has failed that I never heard of*

How can changes be handled?

- Evolution due to environment changes is called *adaptation*
- Evolution and adaptation are traditionally performed *off-line*, but they are increasingly performed *on-line* at run time (see continuously running systems)
- Adaptation can be self-managed (*self-adaptive systems*)

  - J. Kephart, D. Chess, The vision of autonomic computing. IEEE Comput. 2003
  - R. de Lemos et al., Software engineering for self-adaptive systems. Dagstuhl Seminar 2009
  - E. Di Nitto et al., A journey to highly dynamic, self-adaptive service-based applications. ASE Journal, 2008
  - Software Engineering for Adaptive and Self-Managing Systems (SEAMS), starting 2006

The autonomic feedback loop

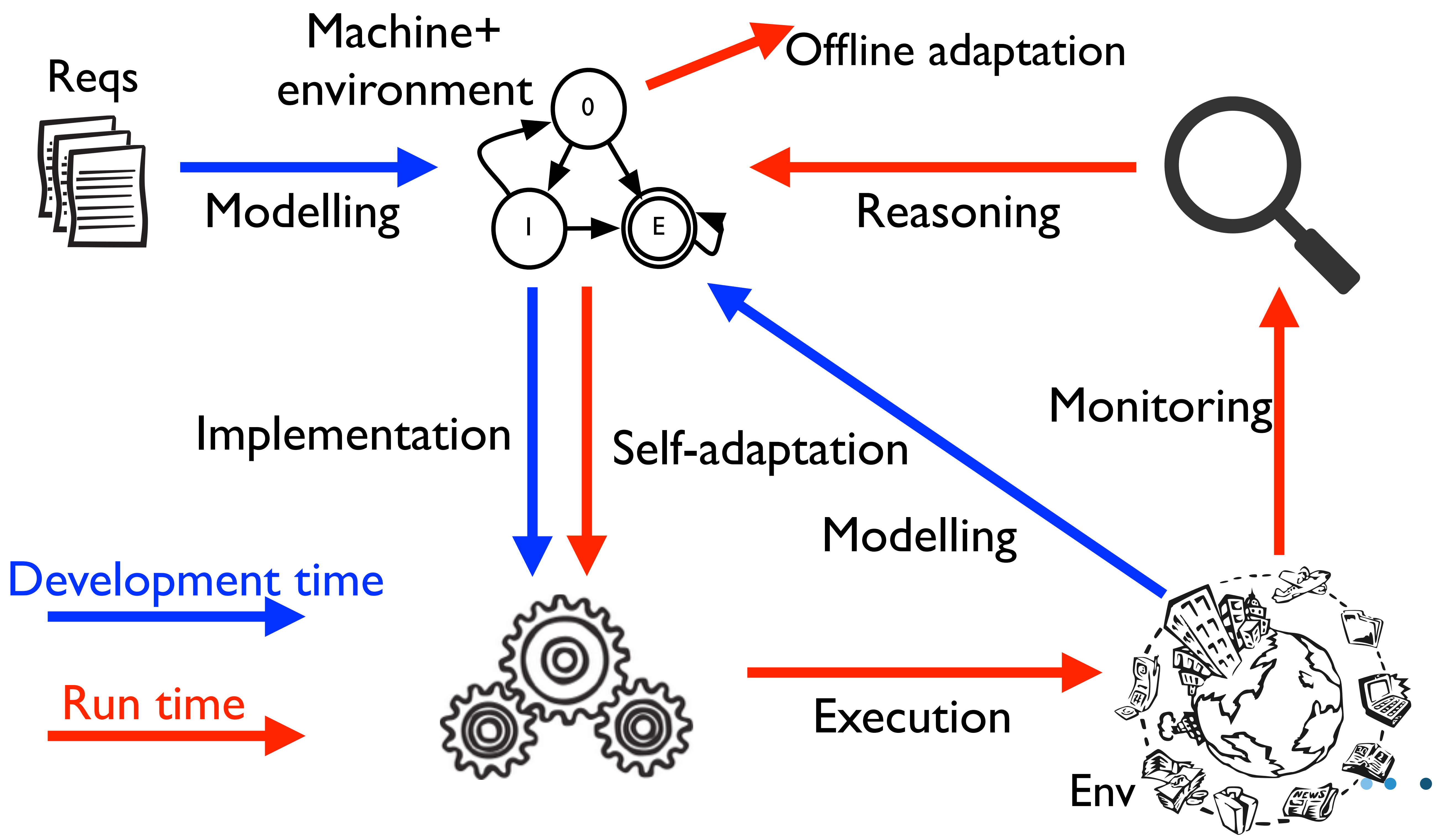


Where are the founding principles?

Paradigm shift

- SaSs ask for a paradigm shift, which involves both development time (DT) and run time (RT)
- The boundary between DT and RT fades
- Reasoning and reacting capabilities must enrich the RT environment
    - detect change
    - reason about the consequences of change
    - react to change

Our view of the lifecycle



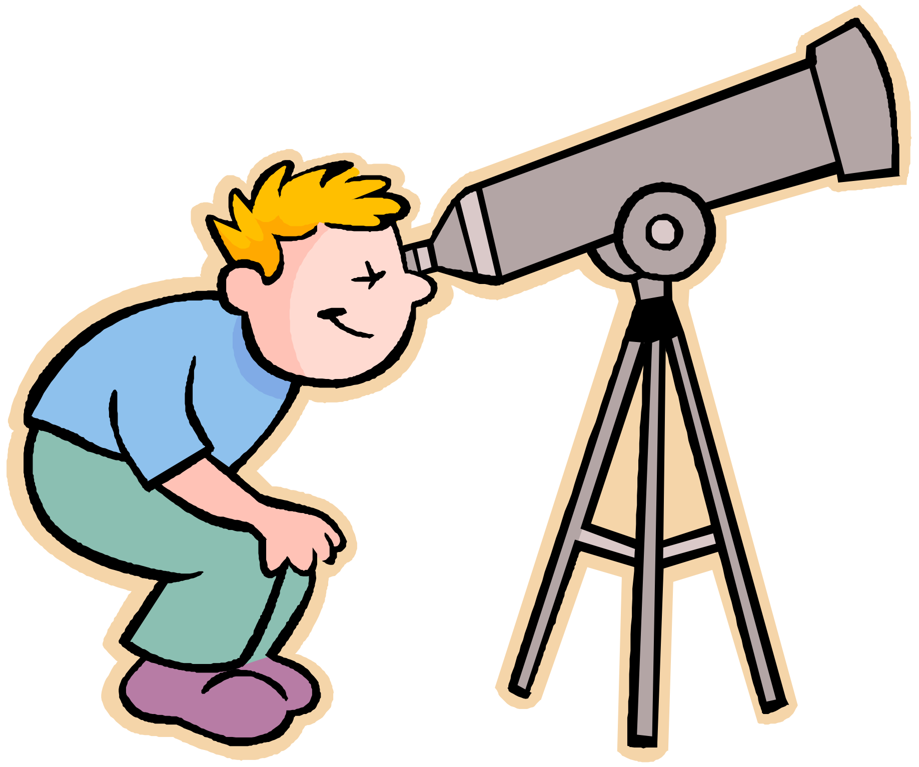
Models&verification@run-time

- To detect change, we need to monitor the environment
- The changes must be retrofitted to models of the machine+environment that support reasoning about the dependability argument (a learning step)
- The updated models must be verified to check for violations to the dependability argument
- In case of a violation, a self-adaptation must be triggered

Known unknowns vs unknown unknowns

- The system can self-adapt to known unknowns
- The unknowns are elicited at design time
- The unknowns become known at run time via monitoring
- If the system has been designed upfront to handle the now knowns, it can self-adapt
- If not, a designer must be in the loop
- There are limits to automation: unknown unknowns cannot even be monitored

Whereof one cannot speak, thereof one must be silent
(Wittgenstein)



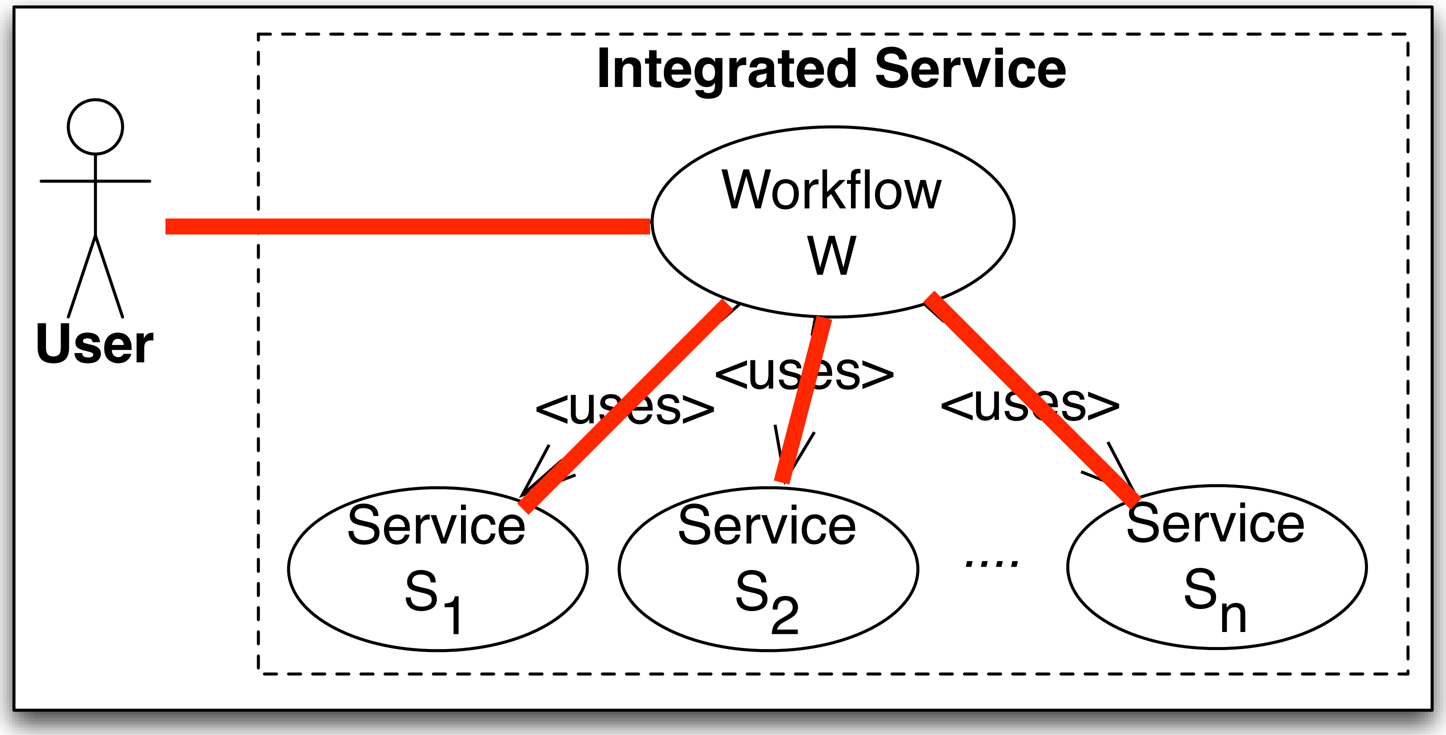
Zooming in

- I. Epifani, C. Ghezzi, R. Mirandola, G. Tamburrelli, "Model Evolution by Run-Time Parameter Adaptation”, ICSE 2009
- C. Ghezzi, G. Tamburrelli, "Reasoning on Non Functional Requirements for Integrated Services”, RE 2009
- A. Filieri, C. Ghezzi, G. Tamburrelli, “ Run-time efficient probabilistic model checking”, ICSE 2011
- A. Filieri, C. Ghezzi, G. Tamburrelli, “Supporting Self-adaptation via Quantitative Verification and Sensitivity Analysis at Run Time, IEEE TSE, January 2016

An exemplary framework

- QoS requirements
  - performance (response time), reliability (probability of failure), cost (energy consumption)

Sources of uncertainty (and change)



Non-functional requirements are quantitative

- Functional requirements are often qualitative ("the system shall close the gate as the sensor signals an incoming train" or "it should never happen that the gate is open and the train is in the intersection")
- Non-functional requirements refer to quality and they are often quantitative ("average response time shall be less than 3 seconds"); often they are probabilistic
- LTL, CTL temporal logics are typical examples of qualitative specification languages
- Non-functional requirements ask for quantitative logics and quantitative verification

Formal modeling and analysis

- S, E can often be formalized via probabilistic Markovian models for non functional rquirements (reliability, performance, energy consumption)

- R formalized via probabilistic temporal logic, e.g. PCTL

- Verification performed via probabilistic model checking

Brief intro to Discrete Time Markov Chains

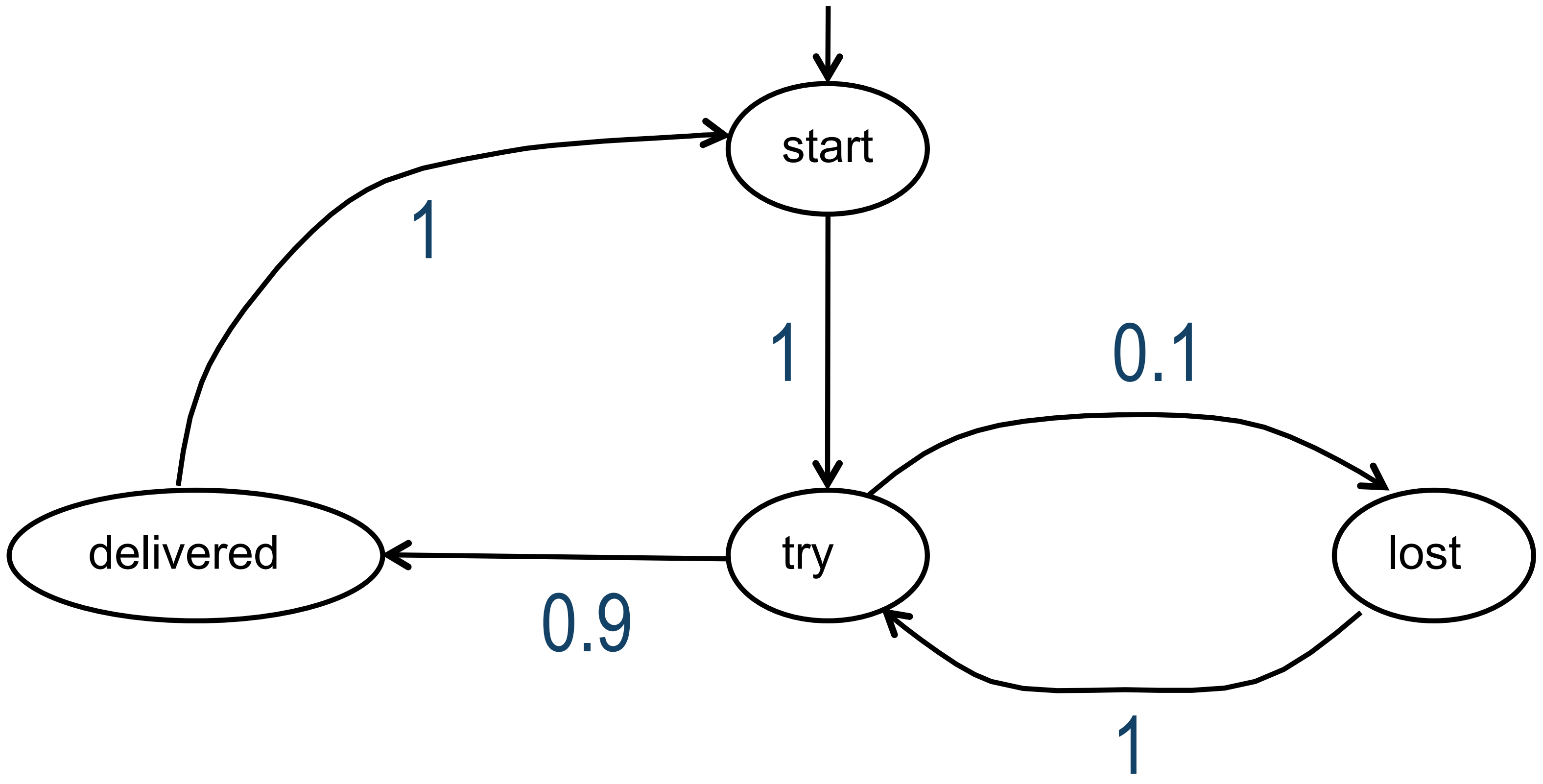
A DTMC is defined by a tuple (S, s_0, P, AP, L) where

- S is a finite set of states
- $s_0 \in S$ is the initial state
- P: $S \times S \rightarrow [0;1]$ is a stochastic matrix
- AP is a set of atomic propositions
- L: $S \rightarrow 2^{AP}$ is a labelling function.

The modelled process must satisfy the Markov property, i.e., the probability distribution of future states does not depend on past states; the process is memoryless

An example

A simple communication protocol operating with a channel



$$
\begin{array}{c|cccc}
 & S & D & T & L \\
\hline
S & 0 & 0 & 1 & 0 \\
D & 1 & 0 & 0 & 0 \\
T & 0 & 0.9 & 0 & 0.1 \\
L & 0 & 0 & 1 & 0
\end{array}
$$

matrix representation

Note: sum of probabilities for transitions leaving a given state equals 1

C. Baier, JP Katoen, “Principles of model checking” MIT Press, 2008

Discrete Time Markov Reward Models

- Like a DTMC, plus
  - states/transitions labeled with a reward
  - rewards can be any real-valued, additive, non negative measure; we use non-negative real functions
- Use in modeling
  - rewards represent energy consumption, average execution time, outsourcing costs, pay per use cost, CPU time

Reward DTMC

- A R-DTMC is a tuple (S, s_0, P, AP, L, μ), where S, s_0, P, L are defined as for a DTMC, while μ is defined as follows:
  - $\mu : S \rightarrow R_{\geq 0}$ is a state reward function assigning a non-negative real number to each **state**
    - ... at step 0 the system enters the initial state s0. At step 1, the system gains the reward μ(s0) associated with the state and moves to a new state...

PCTL

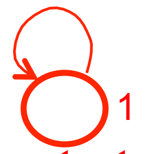
- Probabilistic extension of CTL
- In a state, instead of existential and universal quantifiers over paths we can predicate on the probability for the set of paths (leaving the state) that satisfy property
- In addition, path formulas also include step-bounded until $\phi 1 \; U^{\leq t} \; \phi 2$

$$\Phi ::= \mathsf{true} \mid a \mid \Phi \wedge \Phi \mid \neg \Phi \mid \mathcal{P}_{\bowtie p}(\Psi)$$

$$\Psi ::= X\, \Phi \mid \Phi\, U\, \Phi \mid \Phi\, U^{\leq t}\, \Phi$$

- An example of a reachability property

$P_{>0.8}$ [◊(system state = success)] $\longleftarrow$ **absorbing state**



R-PCTL

Reward-Probabilistic CTL for R-DTMC

$$\Phi ::= \mathsf{true} \mid a \mid \Phi \wedge \Phi \mid \neg \Phi \mid \mathcal{P}_{\bowtie p}(\Psi) \mid \mathcal{R}_{\bowtie r}(\Theta)$$

$$\Psi ::= X\,\Phi \mid \Phi\; U\; \Phi \mid \Phi\; U^{\leq t}\; \Phi$$

$$\Theta ::= I^{=k} \mid C^{\leq k} \mid F\Phi$$

$\mathcal{R}_{\bowtie r}(I^{=k})$ $\mathcal{R}_{\bowtie r}(C^{\leq k})$ $\mathcal{R}_{\bowtie r}(F\Phi)$

true if expected state reward to be gained in the state entered at step k along the paths originating here meets the bound r

true if the expected reward cumulated after k steps meets the bound r

true if the expected reward cumulated before a state satisfying ϕ is reached meets the bound r

Example 1

$$\mathcal{R}_{\bowtie r}(I^{=k})$$

Expected state reward to be gained in the state entered at step k along the paths originating in the given state

"The expected cost gained after exactly 10 time steps is less than 5"

$$\mathcal{R}_{<5}(I^{=10})$$

Example 2

$$\mathcal{R}_{\bowtie r}(C^{\leq k})$$

Expected cumulated reward within k time steps

"The expected energy consumption within the first 50 time units of operation is less than 6 kwh"

$$\mathcal{R}_{<6}(C^{\leq 50})$$

Example 3

$\mathcal{R}_{\bowtie r}(F\Phi)$

Expected cumulated reward until a state satisfying ϕ is reached

“The average execution time until a user session is complete is lower than 150 s”

$$\mathcal{R}_{<150}(F\ \mathbf{end})$$

A bit of theory

- Probability for a finite path $\pi = s_0, s_1, s_2, \ldots$ to be traversed is 1 if $|\pi| = 1$ otherwise $\prod_{k=0}^{|\pi|-2} P(s_k, s_{k+1})$

- A state sj is reachable from state si if a finite path exists leading to sj from si

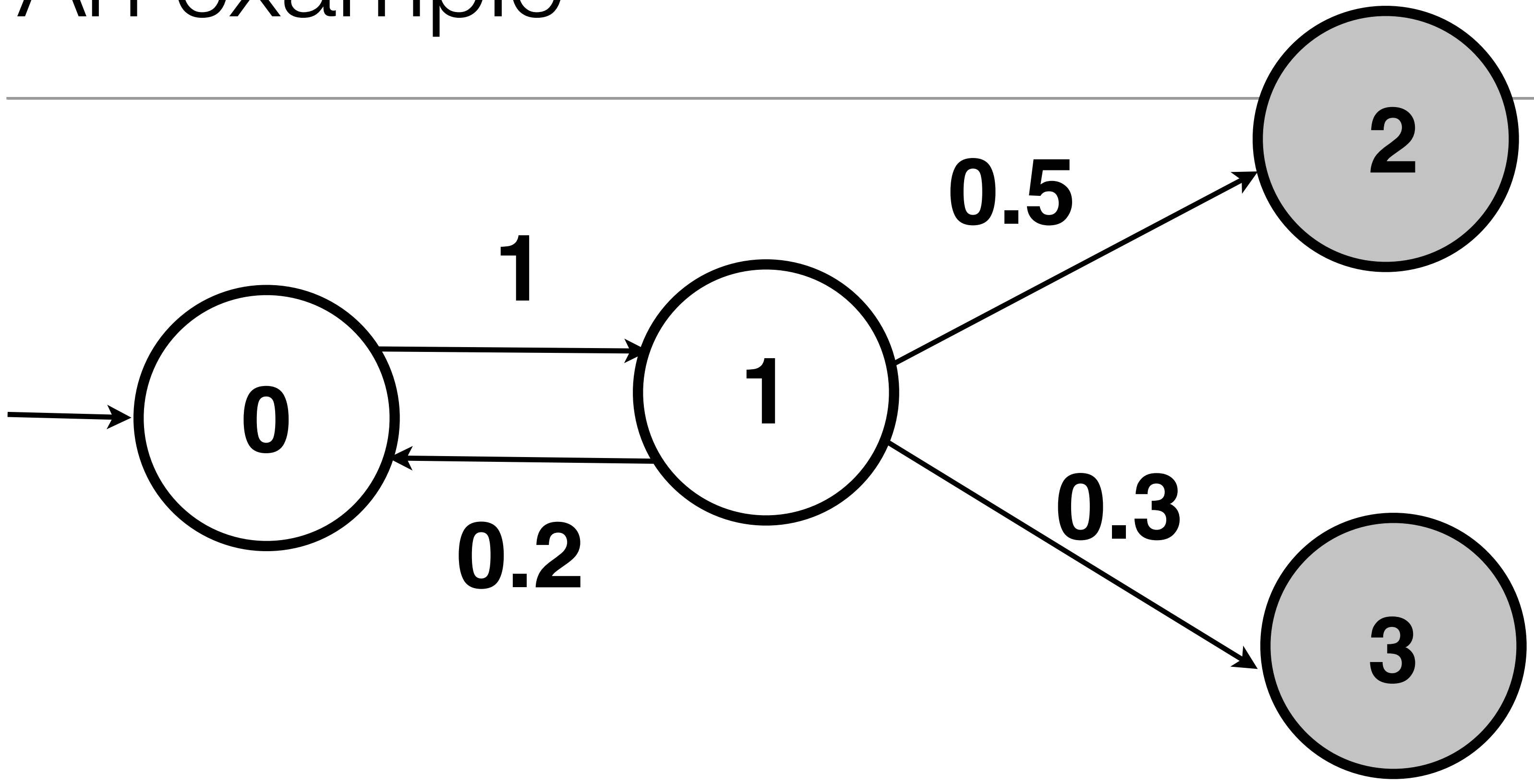
- The probability of moving from si to sj in exactly 2 steps is $\sum_{s_x \in S} p_{ix} \cdot p_{xj}$ which is the entry (i, j) of P^2

- The probability of moving from si to sj in exactly k steps is the entry (i, j) of P^k

A bit of theory

- A state is **recurrent** if the probability that it will be eventually visited again after being reached is 1; it is otherwise transient (a non-zero probability that it will never be visited again)
- A recurrent state s_k where $p_{k, k} = 1$ is called **absorbing**
- Here we assume DTMCs to be **well-formed**, i.e.
    - every recurrent state is absorbing
    - all states are reachable from initial state
    - from every transient state it is possible to reach an absorbing state

An example



$$\begin{pmatrix} \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} \\ \mathbf{0.2} & \mathbf{0} & \mathbf{0.5} & \mathbf{0.3} \\ \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} \end{pmatrix}$$

Probability of reaching an absorbing state (e.g., 2)
2 can be reached by reaching 1 in 0, 1, 2,...∞ steps and then 2 with prob .5

$(1+0.2+0.2^2+0.2^3+ \dots) \times 0.5 = (\sum 0.2^n) \times 0.5 = (1/(1-0.2)) \times 0.5 = 0.625$

Similarly, for state 3, $(1/(1-0.2)) \times 0.3 = 0.375$

Notice that an absorbing state is reached with prob 1

A bit of theory

- Consider a DTMC with *r* absorbing and *t* transient states
- Its matrix can be restructured as
- $$P = \begin{pmatrix} Q & R \\ \mathbf{0} & I \end{pmatrix} \tag{1}$$
  - *Q* is a nonzero *t* × *t* matrix
  - *R* is a t × r matrix
  - 0 is a r × t matrix
  - *I* is a r × r identity matrix

$$Q^k \to \mathbf{0} \text{ as } k \to \infty$$

- Theorem
  - In a well-formed Markov chain, the probability of the process to be eventually absorbed is 1

Reachability properties

- A reachability property has the following form

$$\mathcal{P}_{\bowtie p}(\Diamond\ \phi)$$

  states that the probability of reaching a state where holds matches the constraint
- Typically, they refer to reaching an absorbing state (denoting success/failure for reliability analysis)
- It is a *flat* formula (i.e. no subformula contains $\mathcal{P}_{\bowtie p}(\cdot)$)
- These properties are the most commonly found

A bit on theory

Consider again

$$P = \begin{pmatrix} Q & R \\ \mathbf{0} & I \end{pmatrix} \qquad (1)$$

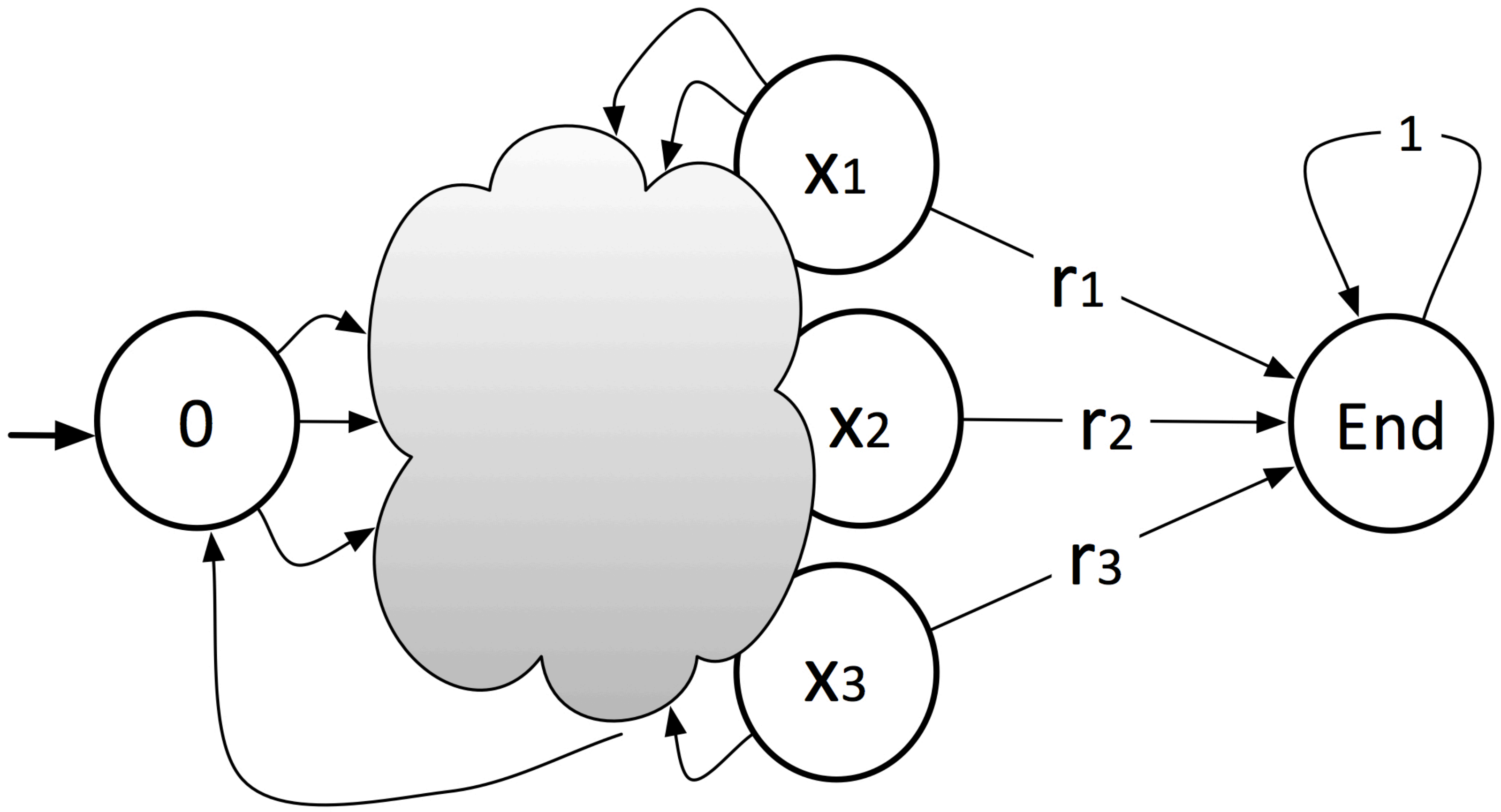
$$N = I + Q^1 + Q^2 + Q^3 + \cdots = \sum_{k=0}^{\infty} Q^k$$

$n_{i,k}$ expected # of visits of transient state s_k from s_i, i.e., the sum of the probabilities of visiting it 0, 1, 2, ...times

Theorem: The geometric series converges to $(I - Q)^{-1}$

Consider $B = N \times R$. The probability of reaching absorbing state s_k from s_i is $b_{ik} = \sum_{j=0..t-1} n_{ij} \cdot r_{jk}$

Proving reachability



$$\Pr(\Diamond s = End) = \sum_j n_{0,j} \cdot r_{j,End}$$

$n_{0,j}$ is the sum of the probabilities to reach state j in 1, 2, 3, ... ∞ steps

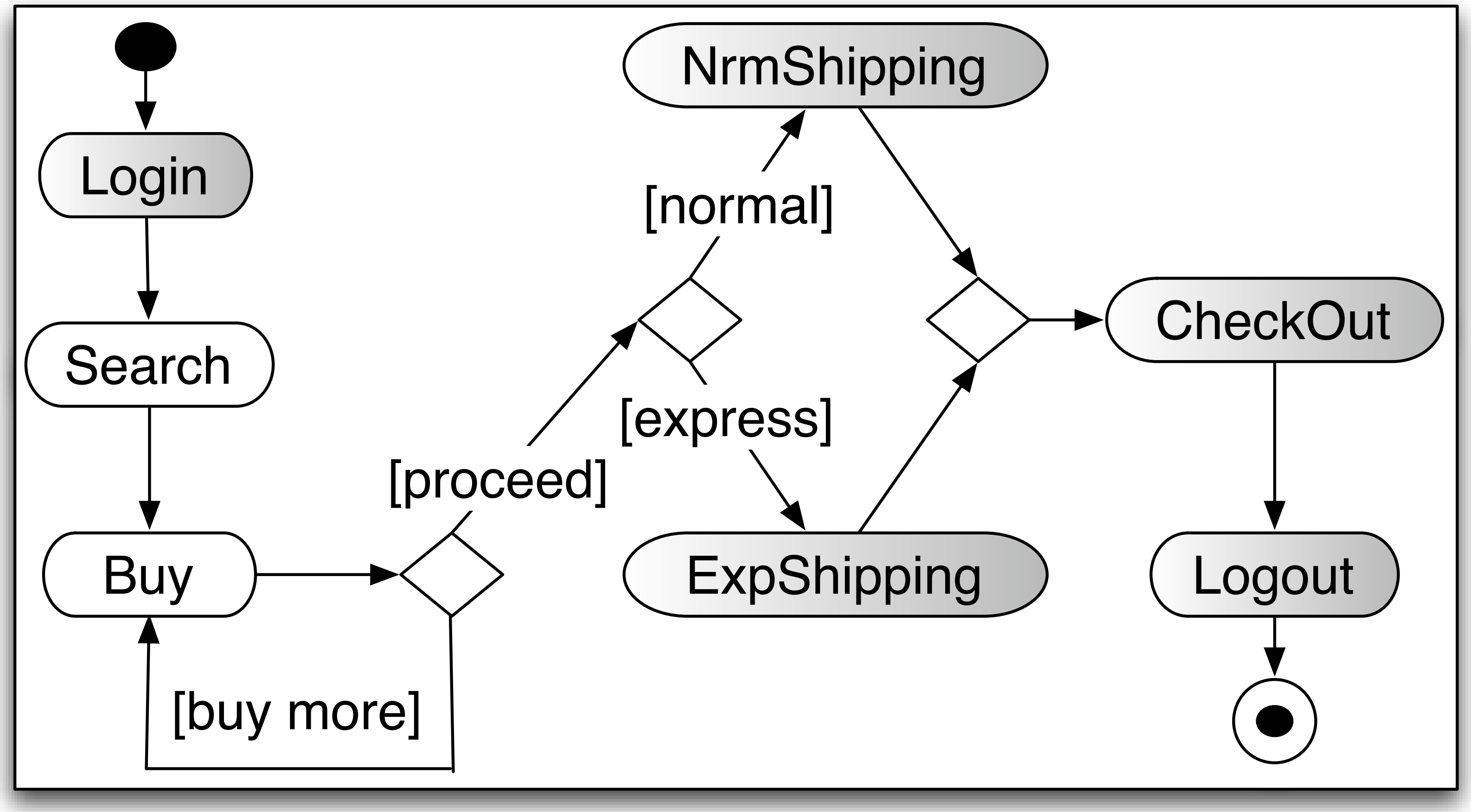
Model checking

- SPIN (Holzmann) analyzes LTL properties for LTSs expressed in Promela
- (Nu)SMV (Clarke et al, Cimatti et al.) can also analyze CTL properties and uses a symbolic representation of visited states (BDDs) to address the “state explosion problem”
- PRISM (Kwiatkowska et al.) and MRMC (Katoen et al.) support Markov models and perform probabilistic model checking

Question

- How can modeling notations and verification fit software evolution?
- Obvious solution:
  - A modification to an existing system viewed as a new system
  - No support to reasoning on the changes and their effects

An e-commerce case study

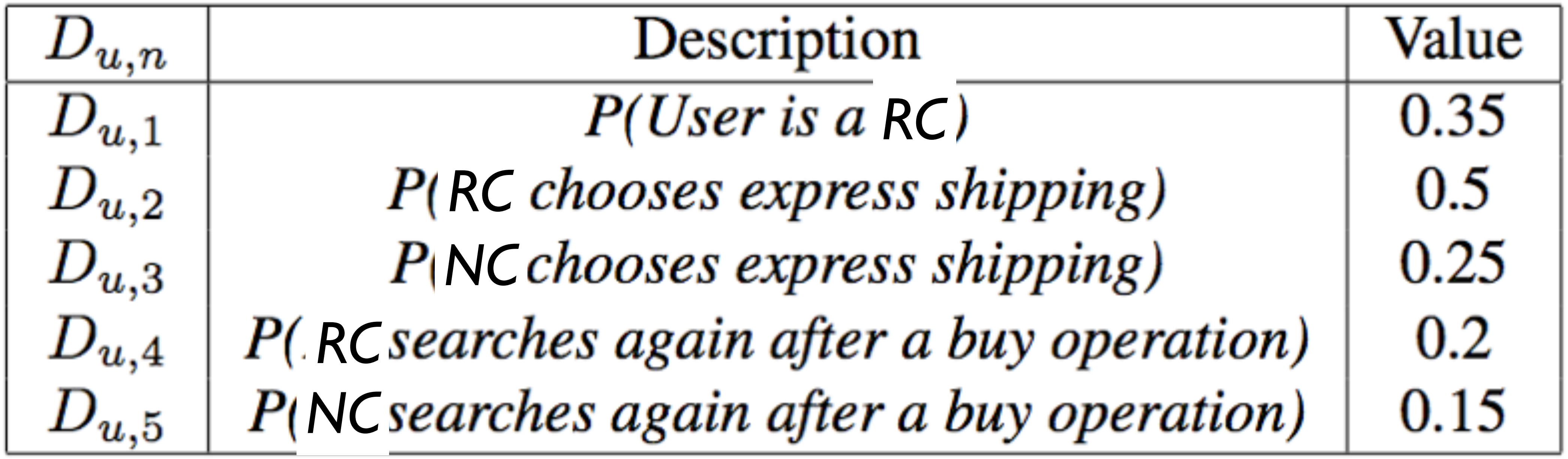


Returning customers
vs
new customers

3 probabilistic requirements:
R1: “Probability of success is > 0.8”
R2:“Probability of a ExpShipping failure for a user recognized as ReturningCustomer < 0.035”
R3:“Probability of an authentication failure is less then < 0.06”

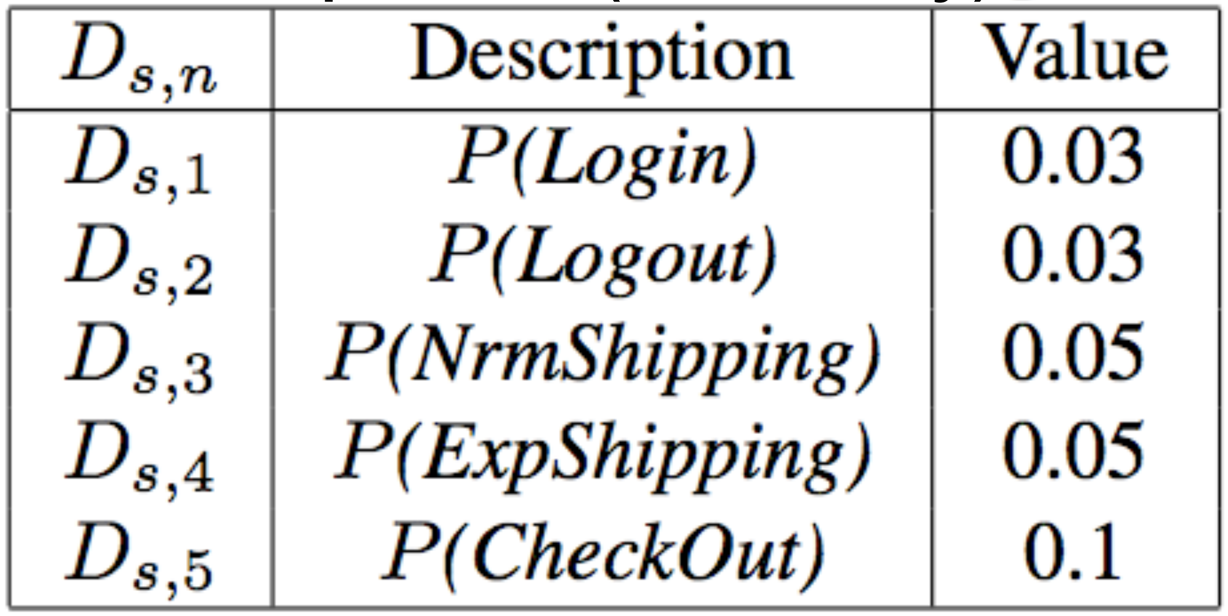
Assumptions

User profile assumptions



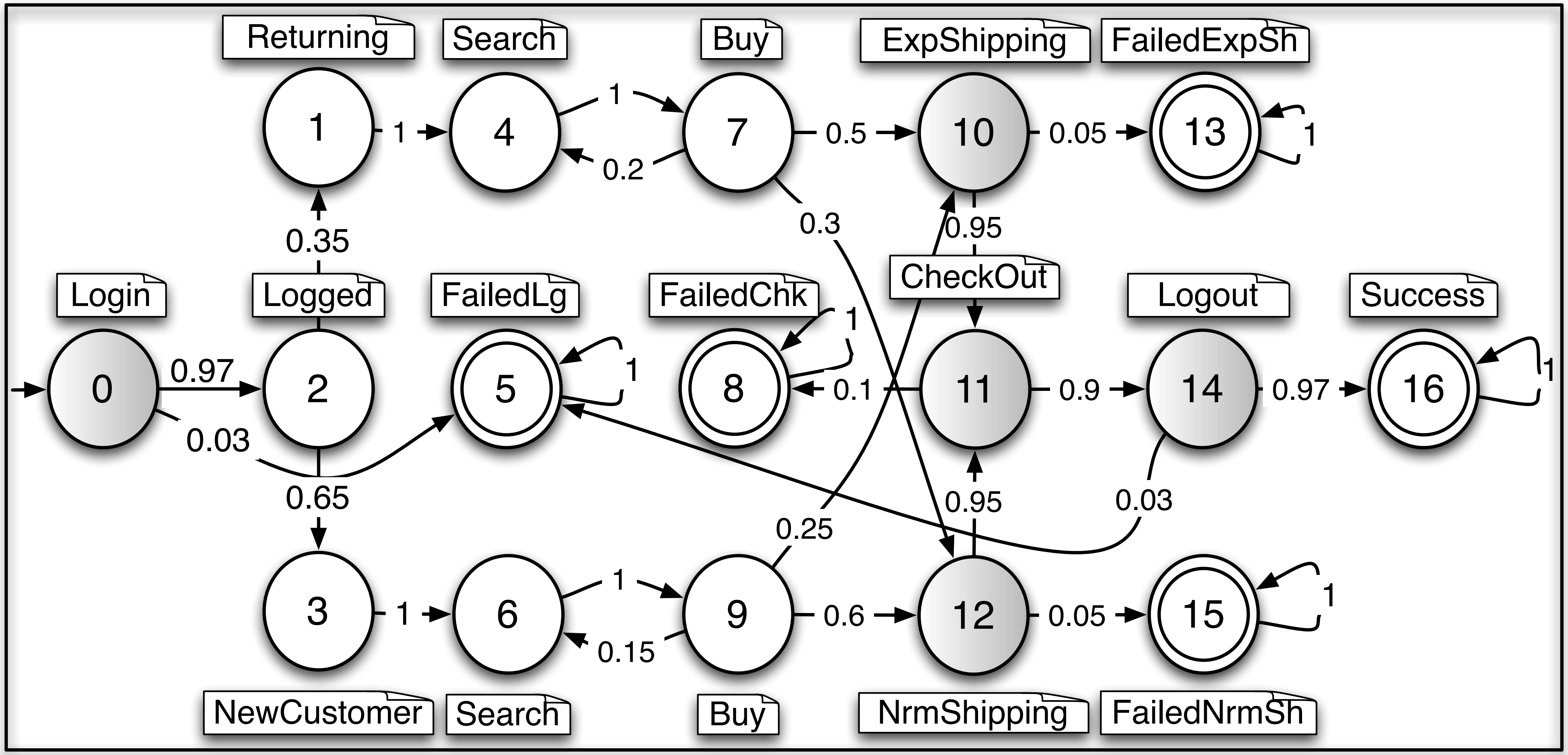
$D_{u,n}$	Description	Value
$D_{u,1}$	*P(User is a RC)*	0.35
$D_{u,2}$	*P(RC chooses express shipping)*	0.5
$D_{u,3}$	*P(NC chooses express shipping)*	0.25
$D_{u,4}$	*P(RC searches again after a buy operation)*	0.2
$D_{u,5}$	*P(NC searches again after a buy operation)*	0.15

External service assumptions (reliability)



$D_{s,n}$	Description	Value
$D_{s,1}$	*P(Login)*	0.03
$D_{s,2}$	*P(Logout)*	0.03
$D_{s,3}$	*P(NrmShipping)*	0.05
$D_{s,4}$	*P(ExpShipping)*	0.05
$D_{s,5}$	*P(CheckOut)*	0.1

S, E, R in practice



R1: Probability of success > 0.8

R2: Probability of ExpShipping failure for ReturningCustomer < 0.035

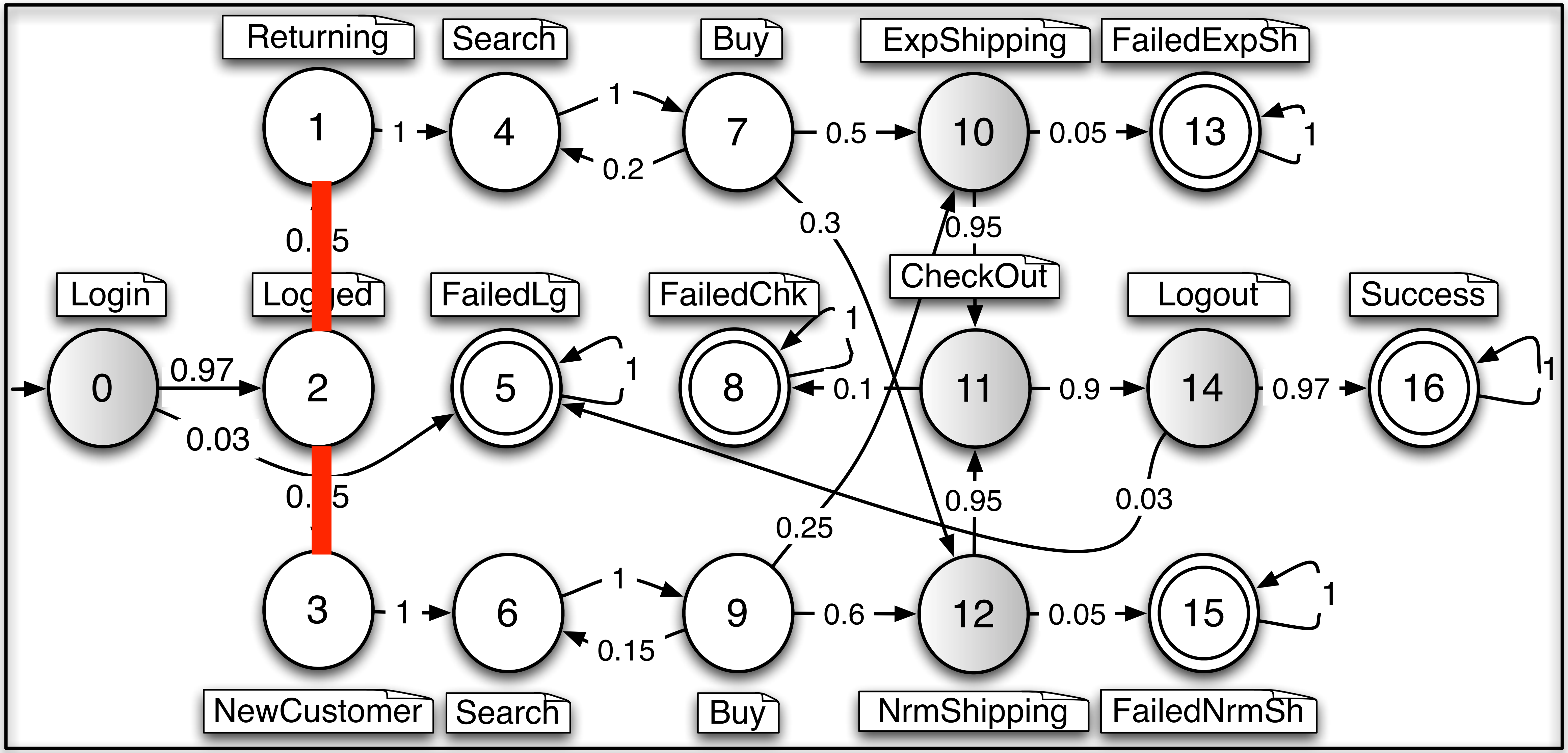
What happens at run time?

- Actual environment behavior is monitored
- Model updated
    - e.g., by using a Bayesian approach to estimate DTMC matrix (posterior) given run time traces and prior transitions

$$\boxed{m_{i,j}^{(N_i)} = \underbrace{\frac{c_i^{(0)}}{c_i^{(0)} + N_i} \times m_{i,j}^{(0)}}_{\text{A-priori Knowledge}} + \underbrace{\frac{N_i}{c_i^{(0)} + N_i} \times \frac{\sum_{h=1}^{d} N_{i,j}^{(h)}}{N_i}}_{\text{A-posteriori Knowledge}}}$$

Verification @ runtime as a trigger for adaptation

- The model has a predictive nature
    - Requirements violation on model predicts future violations
- This may lead to preventive adaptation prior to violations
- Otherwise it leads to self-healing adaptation


Returning
Search
Buy
ExpShipping
FailedExpSh
Login
FailedLg
FailedChk
CheckOut
Logout
Success
NewCustomer
Search
Buy
NrmShipping
FailedNrmSh
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
0.97
0.03
1
0.2
0.5
0.05
0.3
0.95
0.1
0.9
0.97
0.15
0.6
0.25
0.95
0.03
0.05

Models&Verification @ runtime: challenges

- Paradigm change
  - The development-time / run-time boundary fades
- Real-time constraints prevent applicability of current techniques

Towards efficient verification @ runtime

Make verification incremental

- Instead of checking the model after any change, just try to restrict the check to what has changed
    - easier to say than do!

The quest for incrementality

- Is a fundamental ***engineering*** principle

Given a system (model) S, and a set of properties P met by S

Change = new pair S', P' where $S' = S + \Delta S$ and $P' = P + \Delta P$

Let $\prod$ be the proof of S against P

The proof $\prod'$ of P' against S' can be done by just performing a proof increment $\Delta\prod$ such that $\prod' = \prod + \Delta\prod$

Expectation: $\Delta\prod$ easy and efficient to perform

How can we achieve it?

- By construction and change anticipation
    - leveraging modularity and encapsulation
    - assume/guarantee
- By automatic scope detection

An effective technique: incrementality by parameterization

- Requires anticipation of changing parameters, represented as symbolic variable
    - The model is partly numeric and partly symbolic
- Evaluation of the verification condition requires partial evaluation (mixed numerical/symbolic processing)
    - Result is a formula (polynomial for reachability on DTMCs)
- Evaluation at run time substitutes actual values to symbolic parameters and is very efficient

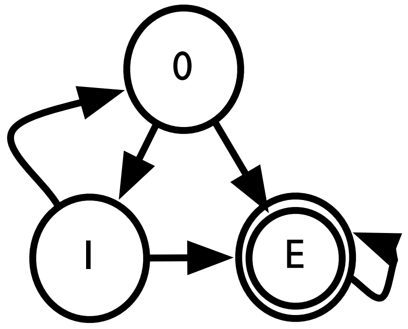
Working mom paradigm

Cook first

Warm-up later

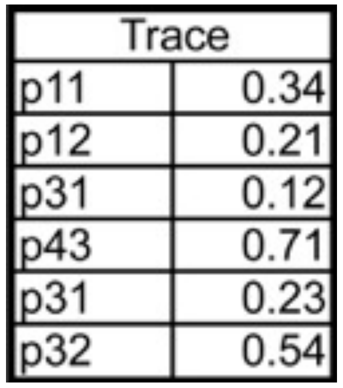
The parametric WM approach

Design-Time
(offline)



Partial
evaluation

Run-Time
(online)



Trace	
p11	0.34
p12	0.21
p31	0.12
p43	0.71
p31	0.23
p32	0.54

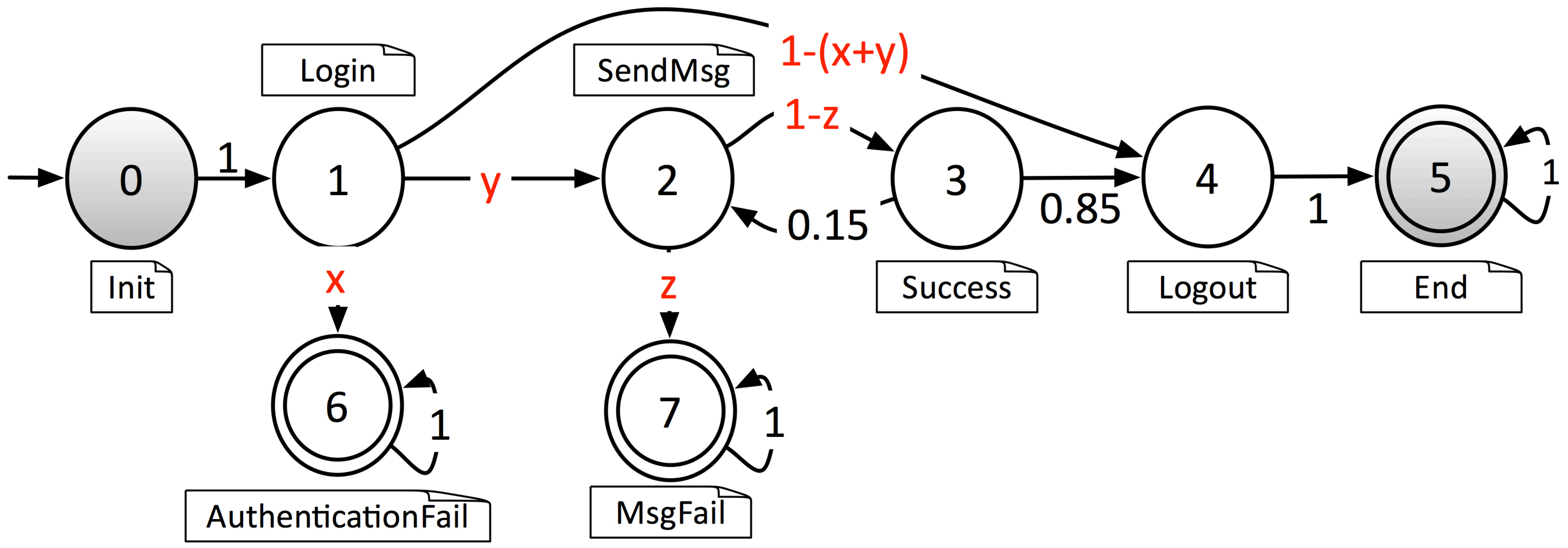
Parameter
values

The approach

- Assumes that the Markov model is well formed
- Works by symbolic/numeric matrix manipulation
- All of (R) PCTL covered
- Does partial evaluation (mixed computation, Ershov 1977)
- Expensive design-time partial evaluation, fast run-time verification
  - symbolic matrix multiplications, but very sparse and normally only few variables

An example

$$r = \Pr(\Diamond\, s = 5) > \bar{r}$$



$$r = \frac{0.85 - 0.85\cdot x + 0.15\cdot z - 0.15\cdot x\cdot z - y\cdot x}{0.85 + 0.15\cdot z}$$

Additional benefit: sensitivity analysis

Back to theory: flat reachability formula

We need to evaluate $Pr(true\ U\ \{s_j \in T\}) = \sum_{s_j \in T} b_{0j}$

where *B* = *N* x *R*; *N is the inverse of I - Q,* $P = \begin{pmatrix} Q & R \\ \mathbf{0} & I \end{pmatrix}$

$$N = I + Q^1 + Q^2 + Q^3 + \cdots = \sum_{k=0}^{\infty} Q^k$$

$n_{i,k}$ expected # of visits of transient state s_k from s_i, i.e., the sum of the probabilities of visiting it 0, 1, 2, ...times

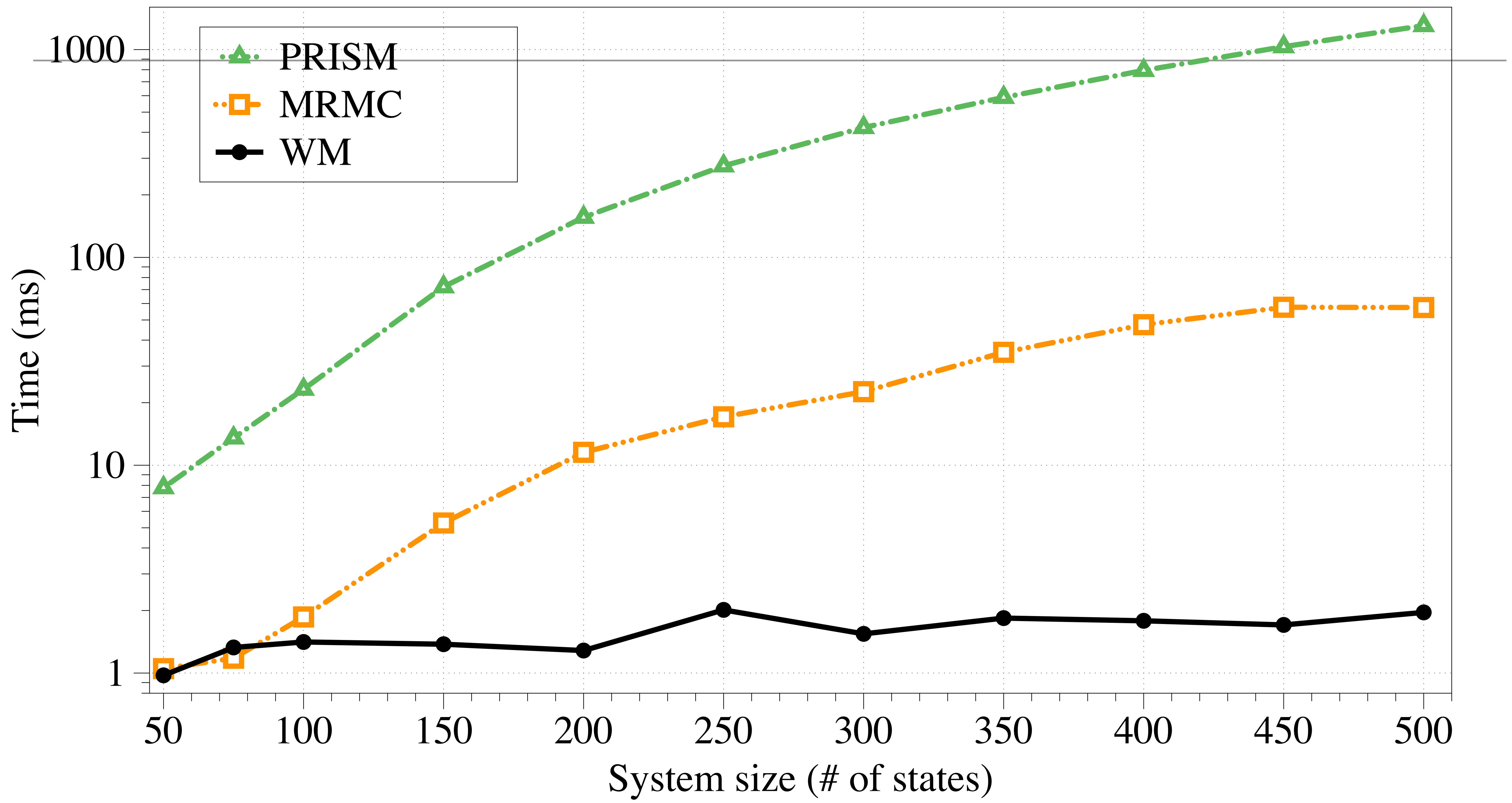
Matrix R is available, we need to compute N

- In our context, N must be evaluated partially, i.e., by a mix of numeric and symbolic processing

Design-time vs run-time costs

- Design-time computation expensive because of numeric/symbolic computations
- Complexity reduced by
    - sparsity
    - few symbolic transitions
    - careful management of symbolic/numeric parts
    - parallel processing
- Run-time computation extremely efficient: polynomial formula for reachability, minor additional complications for full R-PCTL coverage (but still very efficient!)

Run-time performance comparison



128 randomly generated DTMCs, 50 to 500 states (with step 50), two absorbing states, and a normally distributed number of outgoing transitions per state with mean 10 and standard deviation 2. The number of variable states is 4 in each model, thus the number of parameters of each model is normally distributed with mean 40 and standard deviation 8.

Where are we?

- Change is quintessential to software
    - not a nuisance nor something to handle as an afterthought
- Formal methods can set change management on systematic and rigorous grounds that lead to effective and efficient evolution
- They can be brought to runtime to self-manage response to environment changes
- **How can they support a holistic response to changes throughout the software lifetime?**

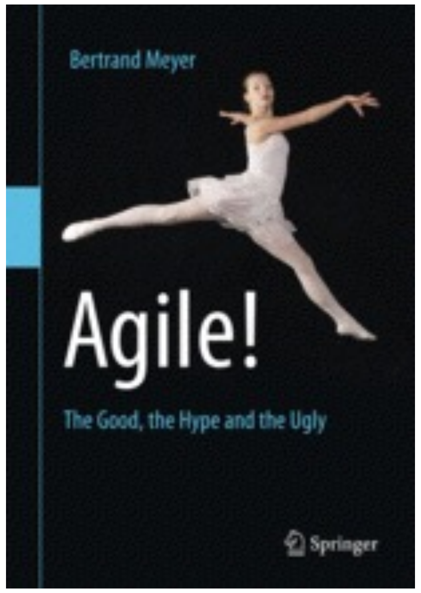
Looking forward: continuous assurance

- Change of perspective: **DevOps**—*the current hype*
    - Development and operation viewed as a continuum
- Focus on assurance that system complies with requirements drives both development and operation
- Focus on continuous assurance requires revisiting verification methods in the light of continuous change

Looking forward

- Software development has become increasingly incremental, change-oriented, ***agile***

B. Meyer, "Agile! The good, the Hype and the Ugly", Springer, 2015



- The good
- The ugly
    - continuous testing: do not wait for a complete system
    - deprecation of upfront activities: requirements (replaced by user stories), specification (replaced by tests), modeling…

- **Get rid of the ugly and move the good one step further**
    - **automate upfront activities and integrate them in agile development**
    - ***can we achieve verification driven development in practice?***

What needs to be done

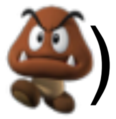
- How can we integrate modelling and verification into iterative, agile development?
  - Support incomplete, partial specifications
    - G. Bruns, P. Godefroid. Model checking partial state spaces with 3-valued temporal logics. In Computer Aided Verification, vol. 1633 LNCS, Springer, 1999.
    - G. Bruns, P. Godefroid. Generalized model checking: Reasoning about partial state spaces. CONCUR 2000
    - M. Chechik, B. Devereux, S. Easterbrook, A. Gurfinkel. Multi-valued symbolic model-checking. ACM TOSEM, 2003.
    - S. Uchitel, G. Brunet, M. Chechik. Synthesis of partial behavior models from properties and scenarios. IEEE TSE 2009.
    - G. Brunet, M. Chechik, D. Fischbein, N. D'Ippolito, S. Uchitel,Weak alphabet merging of partial behaviour models. ACM TOSEM 2011.
  - Support reasoning about changes: support incremental verification

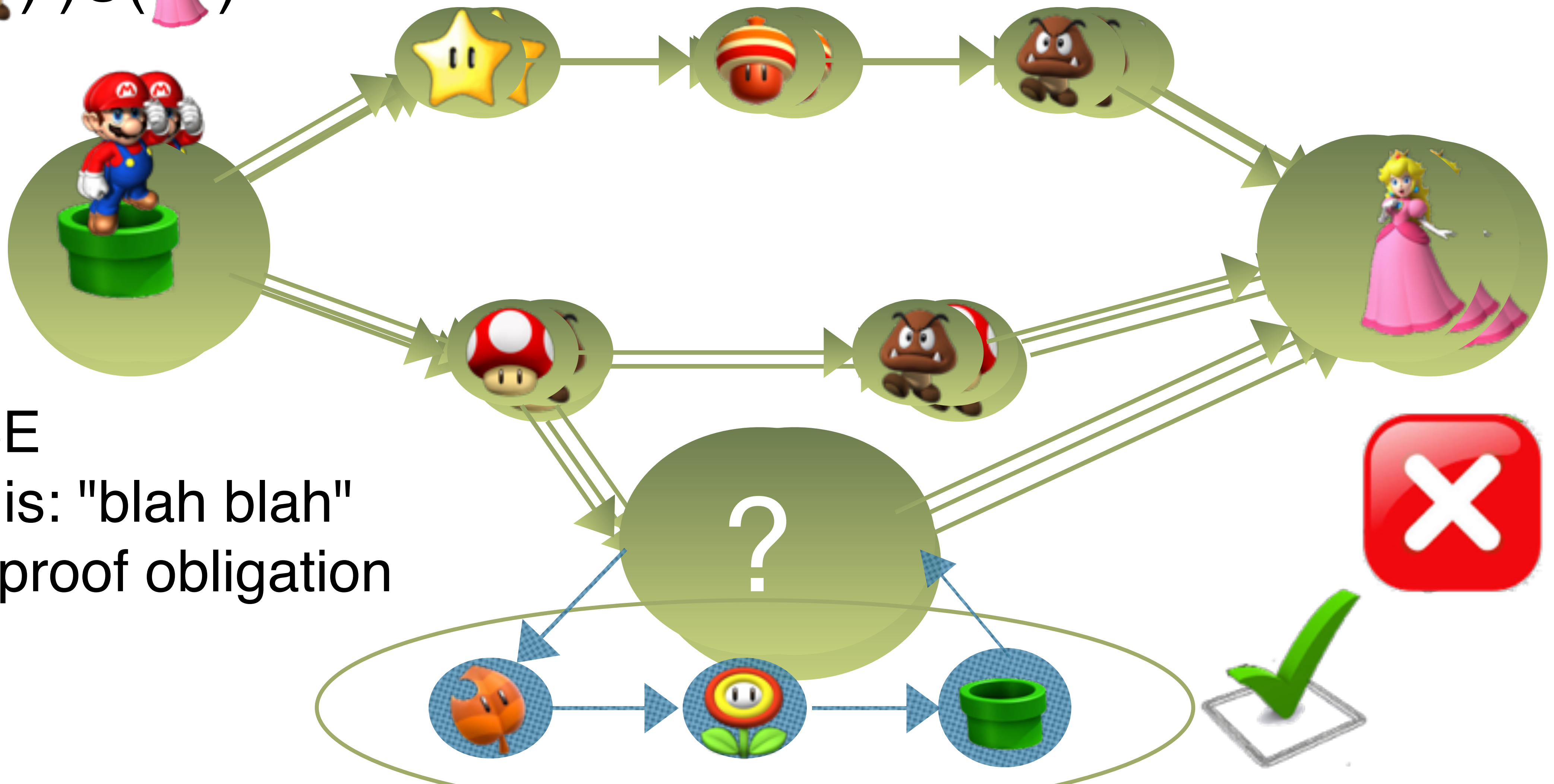
Support to incomplete, partial specifications

- Given an incomplete system (model) S, and a set of properties P to be met by S
- Verification can return YES, NO, MAYBE
- In the MAYBE case, it should compute **proof obligations** for the incomplete parts
- Completion only verifies proof obligations

Example of an incomplete model

- An FSM where certain states stand for an unspecified FSM
  - a functionality whose detail model is postponed

E(¬())U( )



MAYBE
and this: "blah blah"
is the proof obligation

Partial models vs. incremental changes

- Initial decomposition affects the kind of incrementally we get
- Can we achieve incremental verification independent of hierarchical decomposition?
- Can a general approach to incremental verification be found independent of model/program and property language to verify?