Challenges of self-adaptive software

the fading boundary between development time and run time

Internetware 2012 & ISHCS 2012, Qingdao (China)

Carlo Ghezzi
Politecnico di Milano
Deep-SE Group @ DEI-PoliMI

The vision

- **World fully populated by computationally rich devices offering services (disappearing computer)**
  - appliances, sensors/actuators, ... "things"
- **Cyber-physical systems**
- **Mobility**
- **Situation-aware computing**
  - new "services" built dynamically in a situation-dependent manner
- **Continuously running systems**
  - need to evolve while they offer service

The challenge

- Change and flexibility adversary of dependability
- Can they be reconciled through sound design methods?

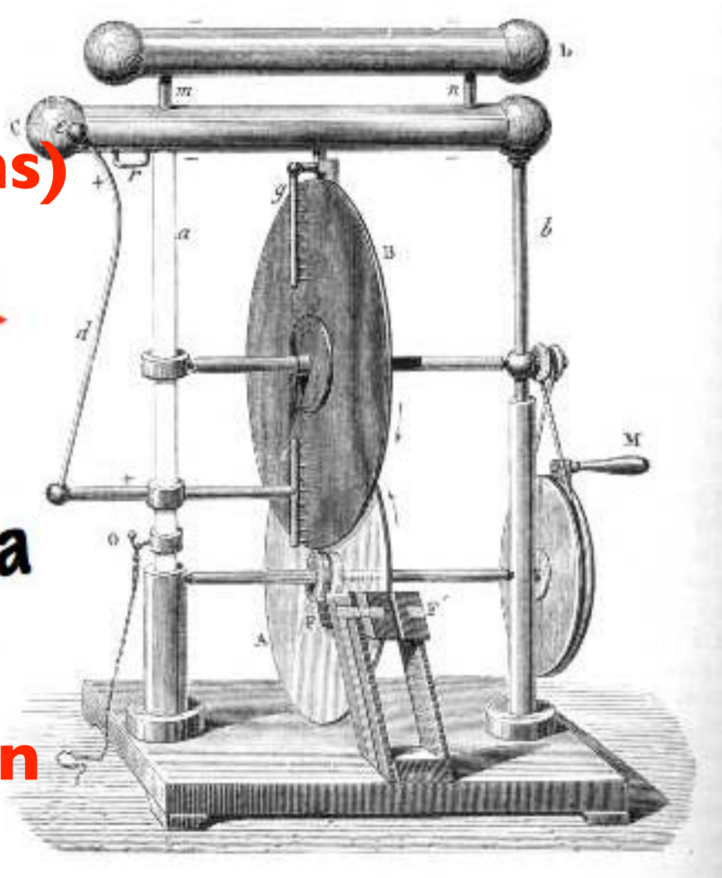
The *machine* and the *world*

World (the environment)

Machine

Domain properties (assumptions)

Shared phenomena



Goals
Requirements

Specification

P. Zave and M. Jackson. Four dark corners of requirements engineering. ACM Trans. Softw. Eng. Methodol., 6(1):1–30, 1997.

Dependability arguments

- Assume that a rigorous (formal) representation is given for
  - R = requirements
  - S = specification
  - D = domain assumptions

if S and D are all satisfied and consistent, it is necessary to prove

- $S, D \models R$

Change comes into play

- Changes in **goals/requirements**
- Changes in **domain assumptions**
  - Usage context
    - request profiles
  - Physical context
    - space, time, …
  - Computational context
    - external components/services (*multiple ownership*)
    - systems increasingly built out of parts that are developed, maintained, and even operated by independent parties
    - no single stakeholder oversees all parts, which may change independently
    - yet by assembling the whole one commits to achieving certain goals

Changes may affect dependability

- Changes may concern
  - R evolution
  - D adaptation *here I focus on D*
- We can decompose D into D_f and D_c
  - D_f is the fixed/stable part
  - D_c is the changeable part

We need to **detect changes to D_c** change detection

and **make changes to S** to keep satisfying R

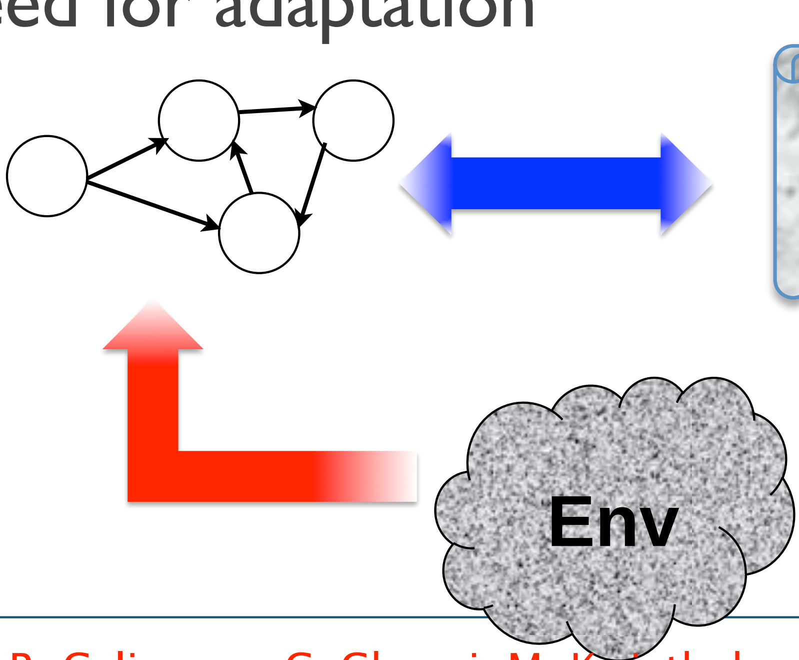
(self) adaptation

Change revisited

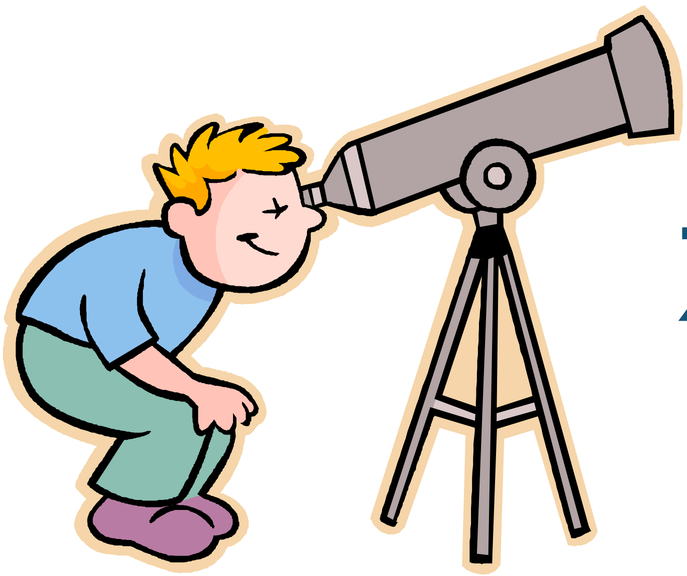
- Change recognized as a crucial problem since the 1970's (see work by M. Lehman)
- Traditionally managed off-line: **software maintenance**
- What is new here
  - the unprecedented degree of change
  - the request that software responds to changes while the system is running (continuously running systems), possibly in a **self-managed** manner

A paradigm change

- Conventional separation between development time and run time is blurring
- Models + requirements need to be kept + updated at run time
- Continuous verification must be performed to detect the need for adaptation



R. Calinescu, C. Ghezzi, M. Kwiatkokwska, R. Mirandola, “Self-adaptive software needs quantitative verification at runtime”, Comm. ACM, Sept. 2012



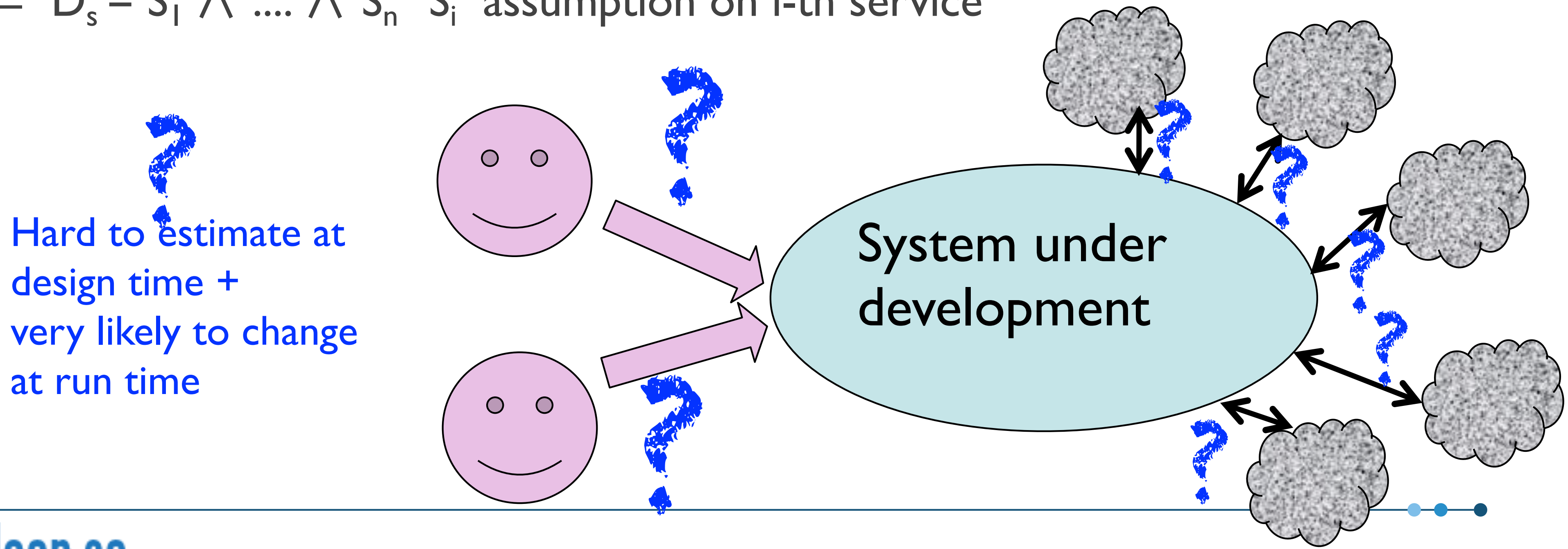
Zoom-in

A framework for (self) adaptation

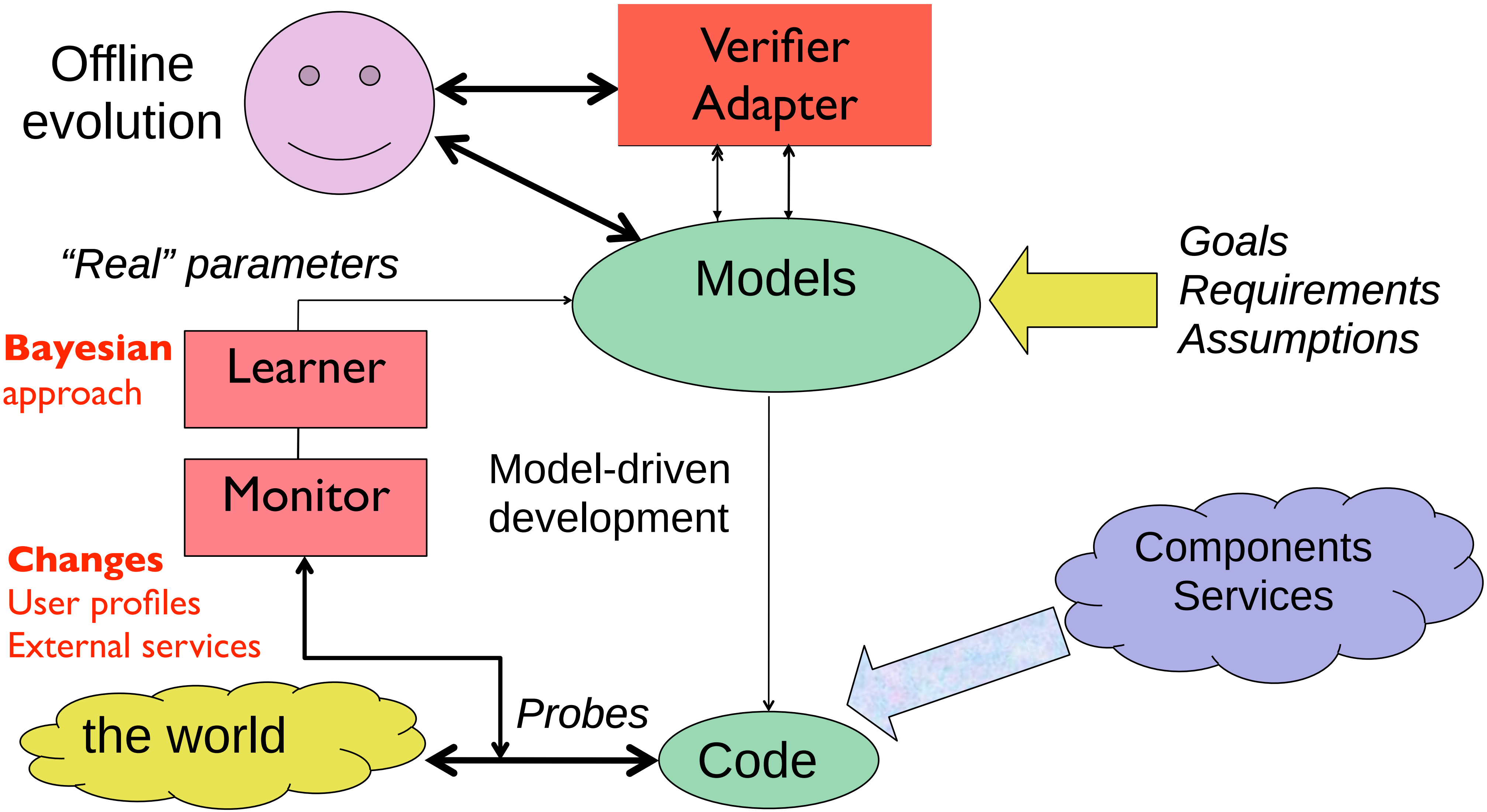
- I. Epifani, C. Ghezzi, R. Mirandola, G. Tamburrelli, "Model Evolution by Run-Time Parameter Adaptation", ICSE 2009
- C. Ghezzi, G. Tamburrelli, "Reasoning on Non Functional Requirements for Integrated Services", RE 2009
- I. Epifani, C. Ghezzi, G. Tamburrelli, "Change-Point Detection for Black-Box Services", FSE 2010

Specific focus

- **Non-functional** requirements
  - reliability, performance, energy consumption, cost, …
- Quantitatively stated in **probabilistic** terms
- D_c decomposed into D_u , D_s
  - D_u = usage profile
  - $D_s = S_l \wedge .... \wedge S_n$ S_i assumption on i-th service



Our approach in a nutshell

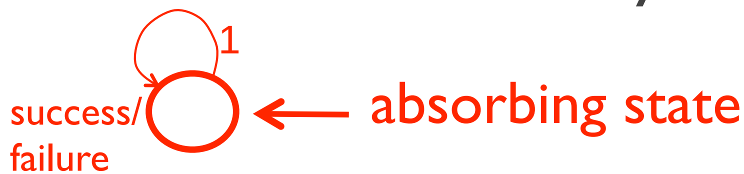


Models

- Different models provide different **viewpoints** from which a system can be analyzed
- Focus on **non-functional** properties and quantitative ways to deal with uncertainty
- Use of **Markov models**
  - DTMCs for reliability
  - CTMCs for performance
  - Reward DTMCs for energy/cost

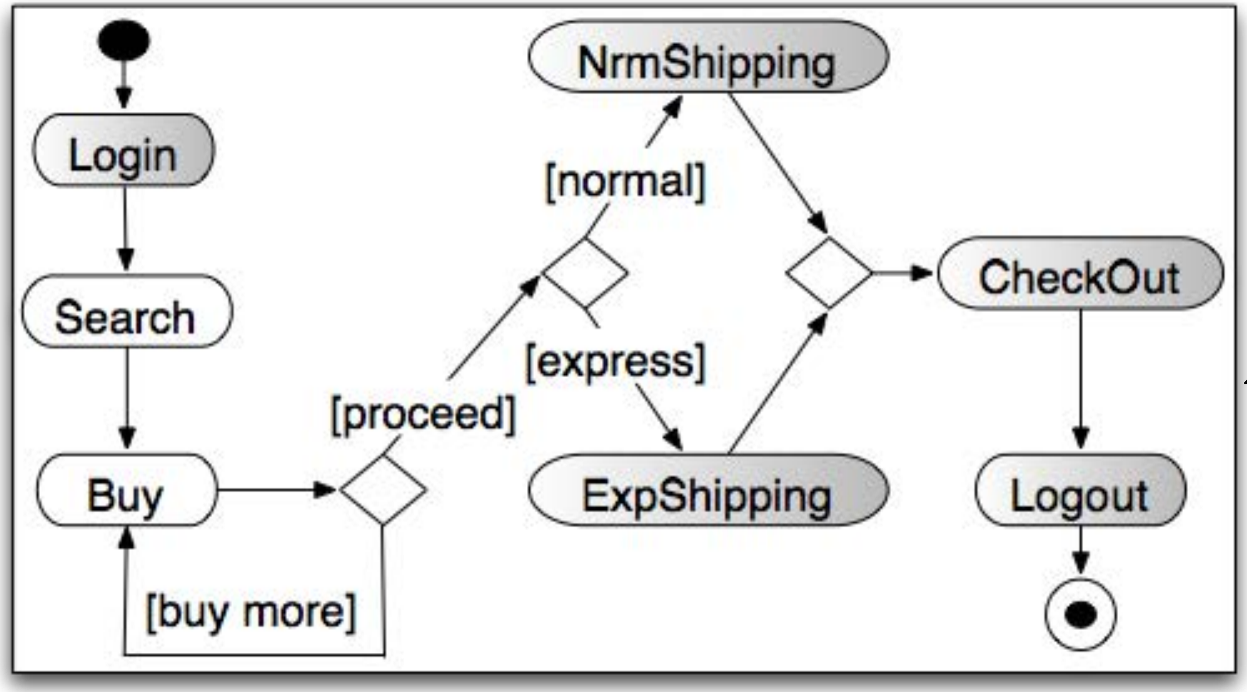
Properties and verification (the case of *reliability*)

- PCTL (probabilistic extension of CTL) provides the necessary expressive power
  - most reliability specifications can be stated as reachability properties
    - $P_{>0.8}$ [$\Diamond$(system state = success)]



- excellent tools exist to perform verification via model checking
  - PRISM (Kwiatkowska et al.)
    - http://www.prismmodelchecher.org/
  - MRMC (Katoen at al.)
    - http://www.mrmc-tool.org/trac/

The approach in in action: e-commerce service composition



Users classified as BigSpender or SmallSpender based on their usage profile.

3 probabilistic requirements:

R1: “Probability of success is > 0.8”

R2:“Probability of a ExpShipping failure for a user recognized as BigSpender < 0.035”

R3:“Probability of an authentication failure is less then < 0.06”

Assumptions

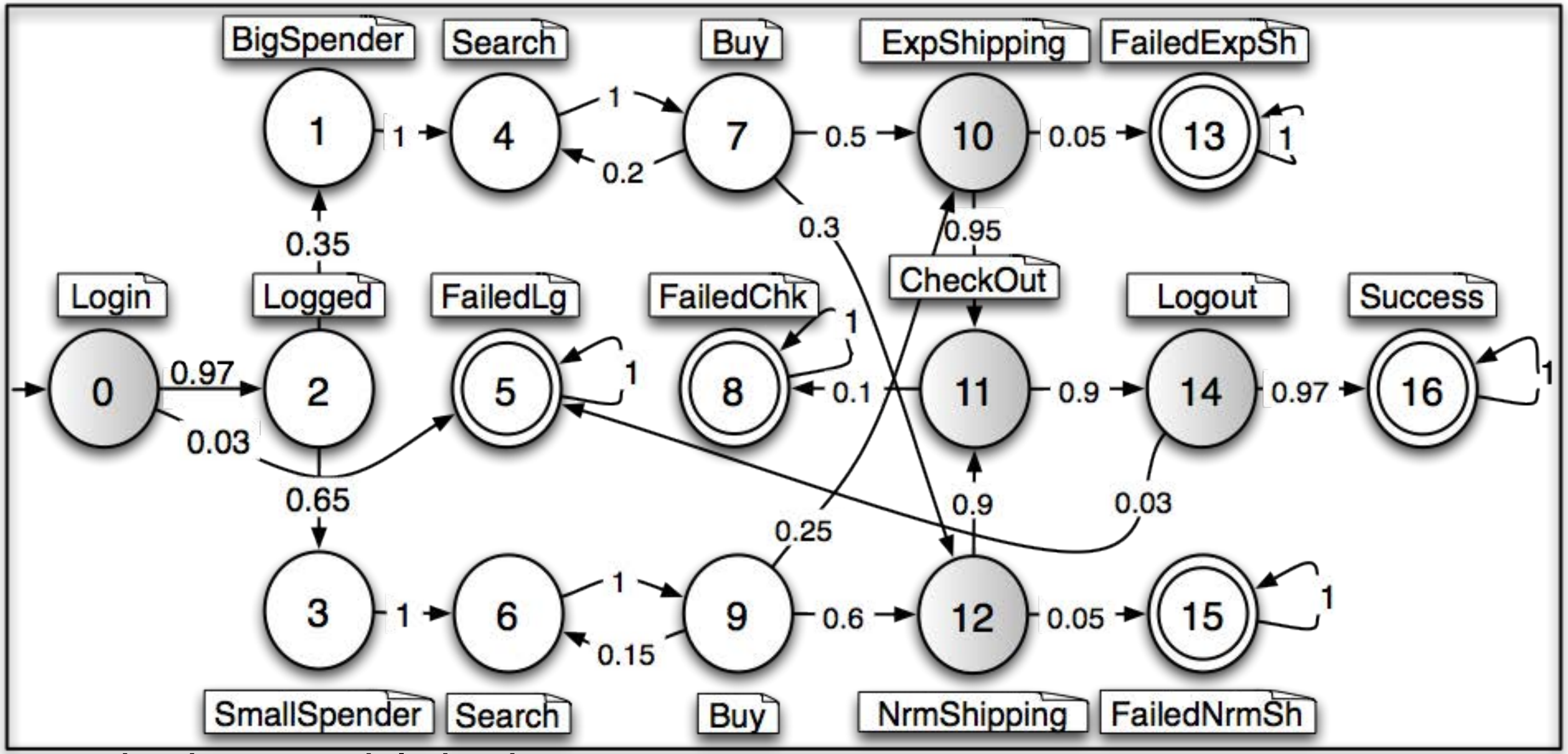
User profile domain knowledge

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

DTMC model



Property check via model checking

R1: "Probability of success is > 0.8" 0.84

R2:"Probability of a ExpShipping failure for a user recognized as BigSpender < 0.035" 0.031

R3:"Probability of an authentication failure is less then < 0.06" 0.056

What happens at run time?

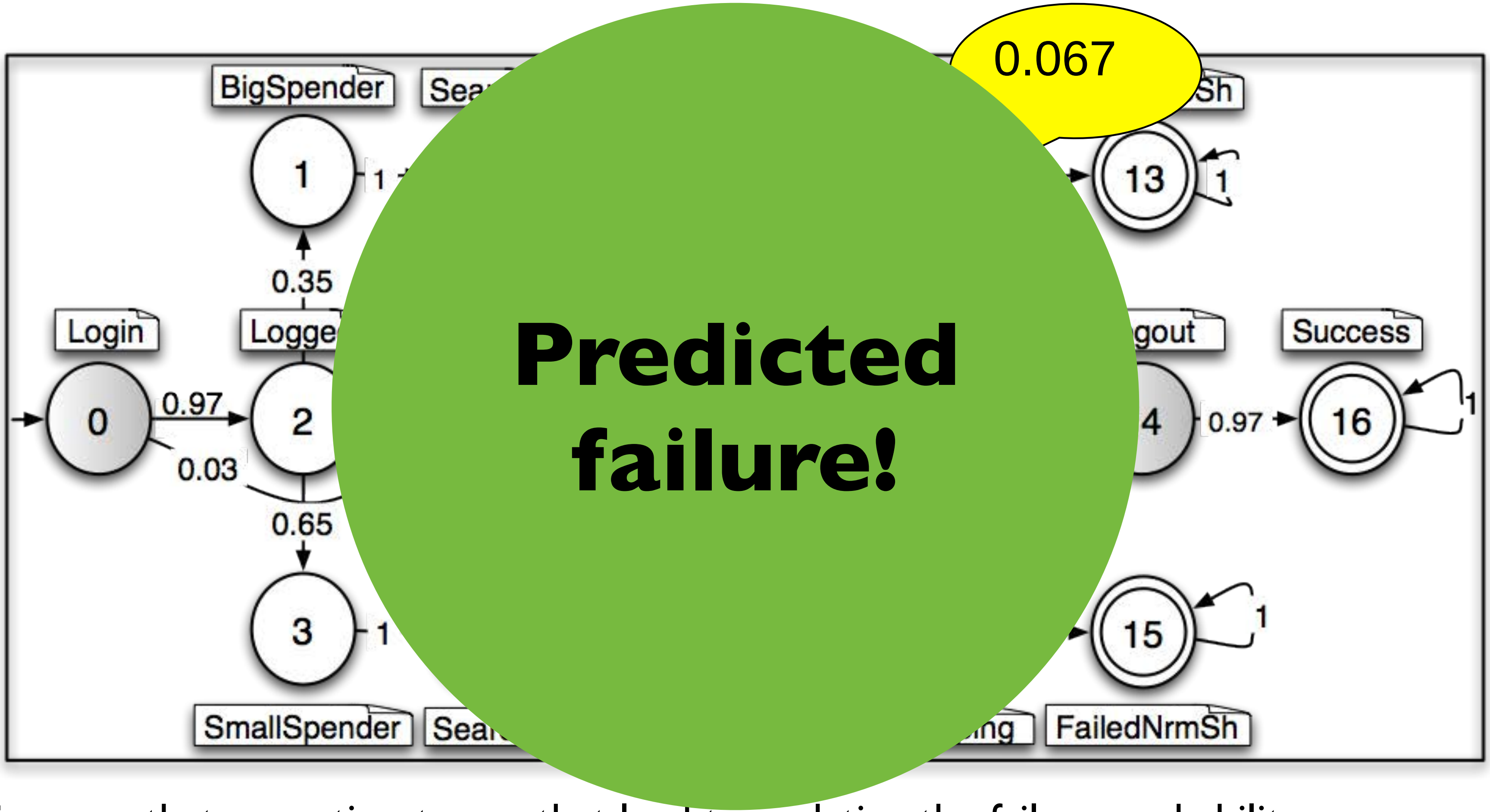
- We monitor the actual behavior
- A statistical (Bayesian) approach estimates the updated DTMC matrix (posterior) given run time traces and prior transitions
- Boils down to the following updating rule

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

Faults and failures

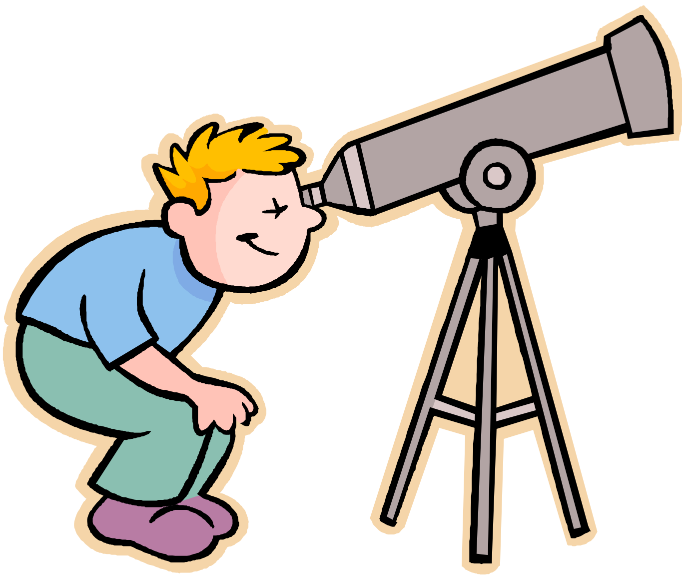
- **Fault**
  - Machine or environment do not behave as expected
- **Failure**
  - Experienced violation of requirement
- Assume that an *environment* fault is detected
  Three cases are possible
  - All Reqs still valid
    - **OK, but my signal contract violation**
  - Some Req violated + violation experienced in real world
    - **Failure detection**
  - Some Req violated, but violation not experience yet
    - **Failure prediction**

Predicted vs. detected failure



Suppose that execution traces that lead to updating the failure probability of ExpShipping are those involving small spenders

R2: "Probability of a ExpShipping failure for a user recognized as BigSpender" < 0.035



Zoom-in
Run-time efficient model checking

A. Filieri, C. Ghezzi, G. Tamburrelli, "Run-Time Efficient Probabilistic Model Checking", ICSE 2011

A Filieri, C. Ghezzi, G. Tamburrelli,, "A formal approach to adaptive software: continuous assurance of non-functional requirements", Formal Aspects of Computing, vol. 4, n. 2, 2012

C. Ghezzi, "Evolution, Adaptation, and the Quest for Incrementality", in Large-Scale Complex IT Systems. Development, Operation and Management, 17th Monterey Workshop 2012, Oxford, UK, March 19-21, 2012, LNCS, Springer 7539

Rethinking run-time environments

- Traditionally software engineering has been mostly concerned with development time
- The result is **code** that simply needs to be **run**

(Self-)adaptive software requires much more

- *must be able to reason at run time about itself and the environment*
  - ✓ *models*
  - ✓ *goals and requirements*
  - ✓ *strategies*

  must be available at runtime

Run-time agility, incrementality

- Agility taken to extremes
  - time boundaries shrink
    - ✓ constrained by real-time requirements
- Verification approaches must be re-visited
  - they must be **incremental**

Given S system (model), P property to verify for S
Change = new pair S', P'

Incremental verification reuses part of the proof of S against P to verify S' against P'

How to make verification incremental

Incrementality by encapsulation

- Grounded on seminal work of D. Parnas (1972)
  - Design for change
    - ✓ changes must be anticipated and encapsulated within *modules*
    - ✓ interface vs implementation
    - ✓ interfaces formalized via contracts (B. Meyer)
- Known as *assume-guarantee* when contextualized to verification (C. Jones)

D.L. Parnas, On the criteria being used to decompose modules into systems, Comm ACM, 1972
B Meyer, Applying “design by contract”, Computer, 1992
C. Jones. Tentative steps toward a development method for interfering programs.ACM TOPLAS, 1983

Assume-guarantee

- Show that module M1 guarantees property P1 assuming that module M2 delivers property P2, and vice versa
- Then claim that the system composed of M1 || M2 guarantees P1 and P2 unconditionally
  - these arguments support *compositional reasoning*
- Approach works if changes do not percolate through the module's interface, affecting contract
  - effect of change *encapsulated* within the boundaries predicted at design time

How to make verification incremental

Incrementality by parameterization

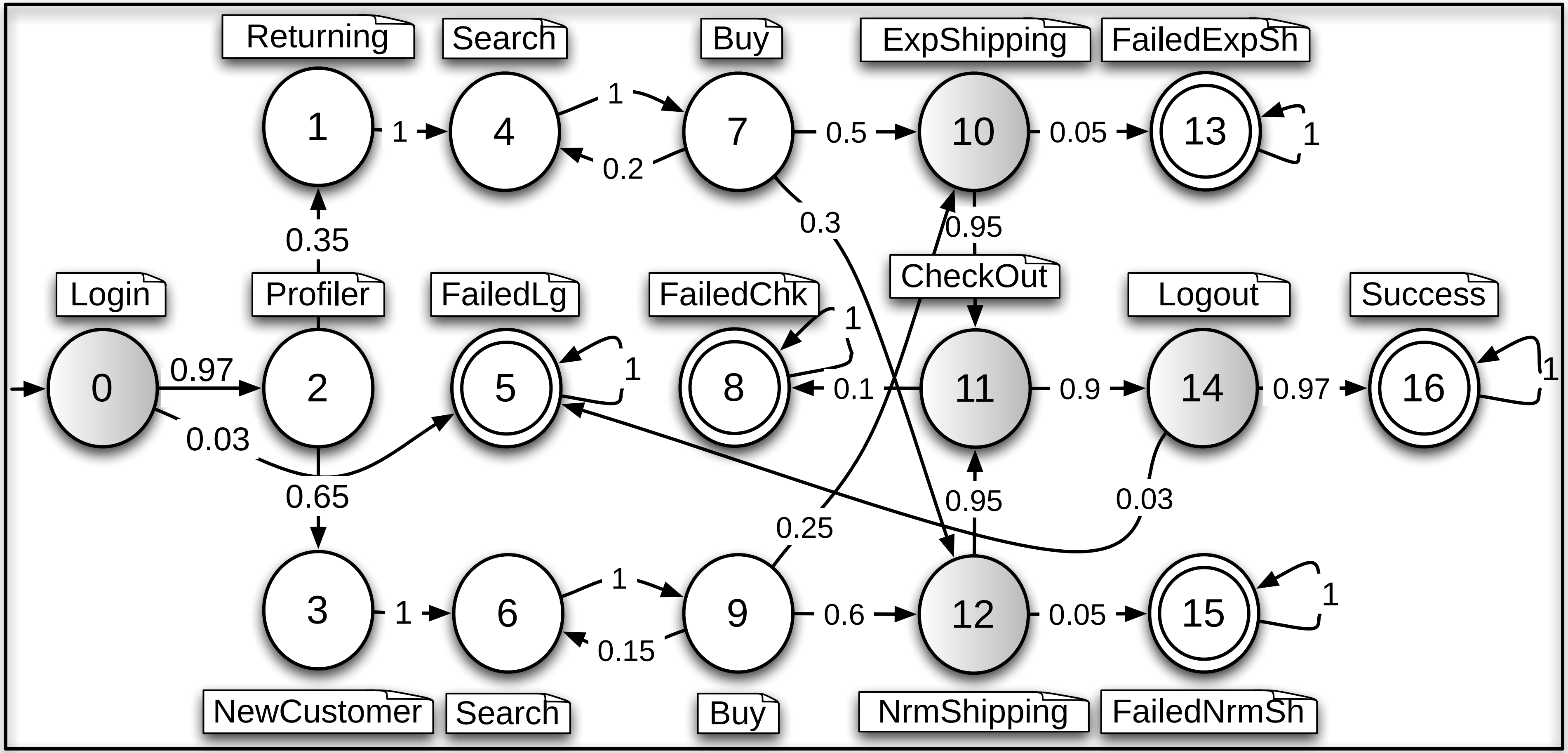
- Requires anticipation of changes, which become parameters
- Does not require modular reasoning
- Still requires identification of elementary sources of change
- Inspired by the concept of partial evaluation (Ershov 1977)

Let P be a program $P : I \rightarrow O$

Suppose I can be partitioned into Is and Id, where Is set of input data known statically, before runtime

Partial evaluation transforms P into an equivalent *residual* program P': $Id \rightarrow O$ from by precomputing static input before runtime

An example



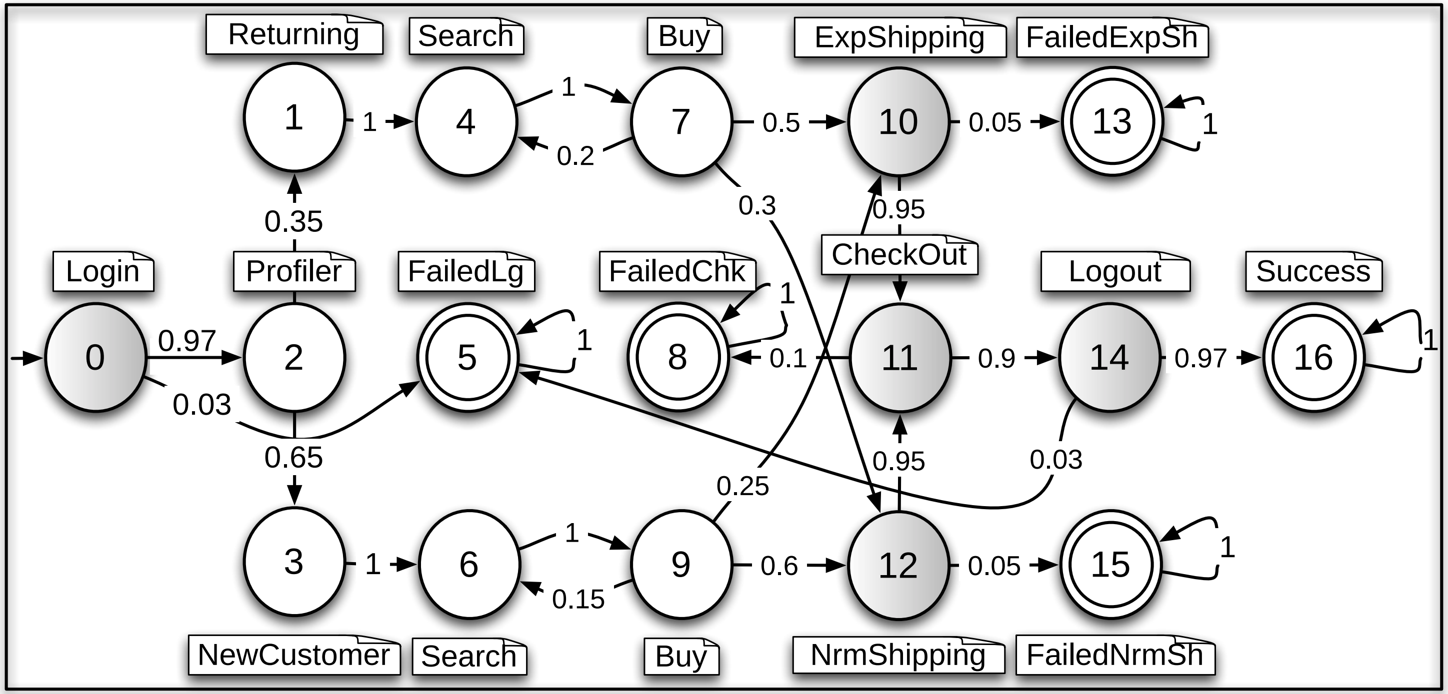
Requirement: $\mathbf{F_{>0.8}\,[\,s = 16]}$

Self Managing Situated Computing

An example (continues)

Satisfaction of requirement **$F_{>0.8}$ [s = 16]** can be checked at design time, but at run-time, e.g. user profiles may change

We can treat them as **variables** and compute at design time a parametric verification formula which is then evaluated at run time



How to make verification incremental

Syntax-driven incrementality

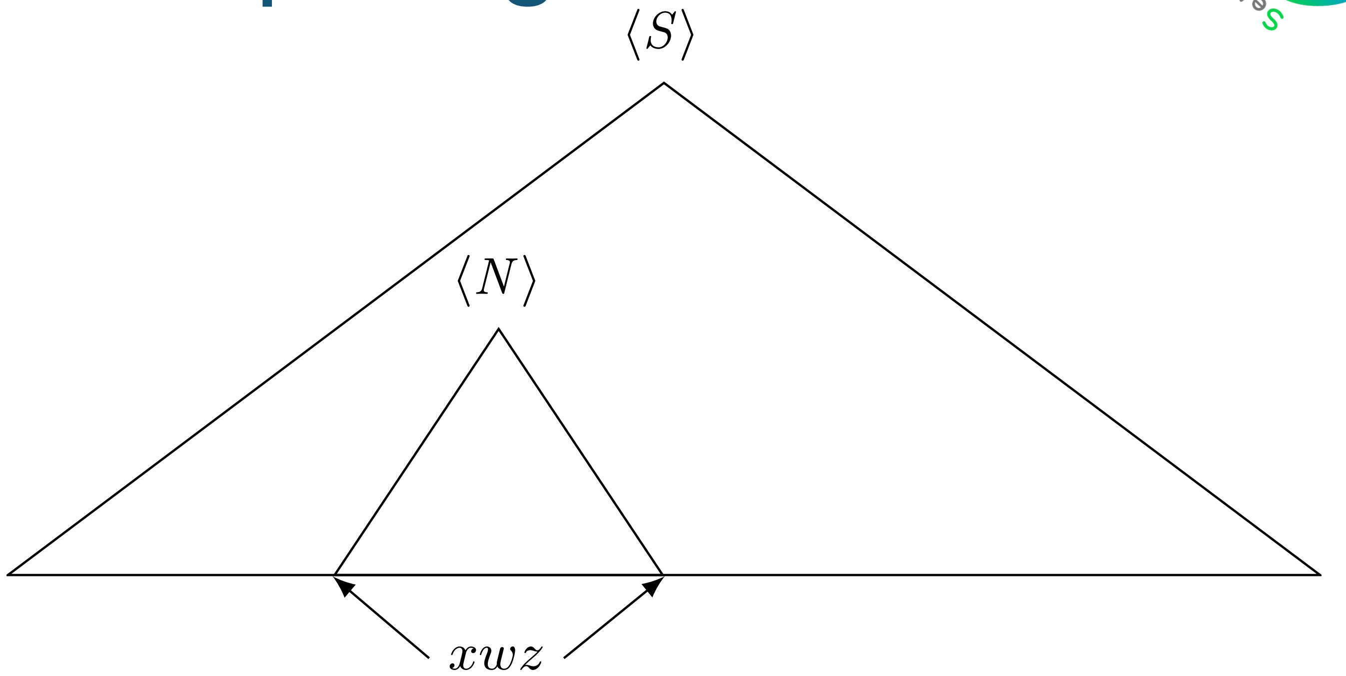
- Assumes artifact to analyze with a syntactic structure expressible as a formal grammar
- Verification is expressed via attributes (à la Knuth)
- Changes can be of any kind

Intuition

Syntax-driven incrementality

- Incremental parsing strategy finds boundary for artifact re-analysis
- Knuth proved that attributes can be only synthesized (computed bottom-up) and thus only need to be recomputed for the changed portion + propagated to the root node

Incremental parsing: intuition



- Assume *w* is the modified portion
- Ideally, re-analyze only a sub-tree “covering” w, rooted in <N>, and “plug-it-in” the unmodified portion of tree
- The technique works if the sub-tree is small, and complexity of re-analysis is the same as complexity of “main” algorithm

Incremental parsing: past and new results

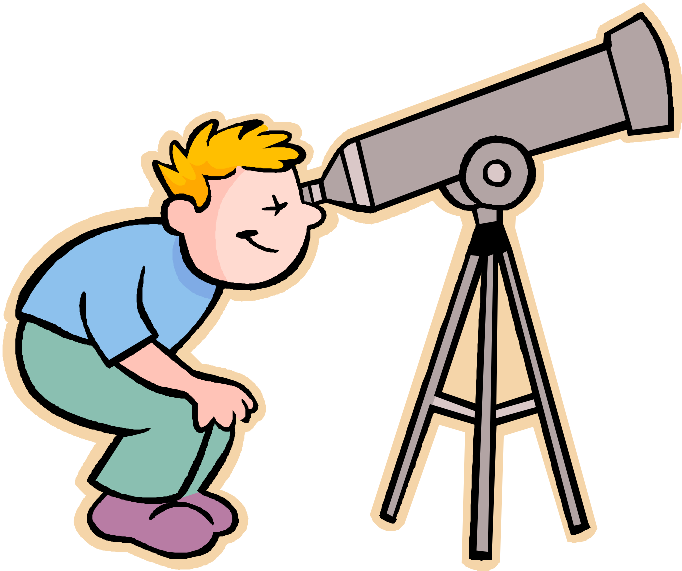
- Past work on "mainstream" LR grammars
  - C. Ghezzi and D. Mandrioli, Incremental parsing, ACM Trans. Program. Lang. Systems, 1979
    - ✓ Saves the maximum possible portion of the syntax tree, but the re-analyzed portion can still be large in certain cases
- Recent work resurrected Floyd's operator precedence grammars
  - R.W. Floyd. Syntactic analysis and operator precedence, Journal of the ACM, 1963
  - S. Crespi Reghizzi and D. Mandrioli. Operator-precedence and the visibly pushdown property, J. Comput. Syst. Sci., to appear.
  - Floyd's grammars cannot generate all deterministic CF languages
  - but in practice any programming language can be described by a Floyd grammar
  - parsing can be started from any arbitrary point of the artifact to be analyzed

Initial validation of the approach

- Case 1: reliability (QoS) analysis of composite workflows
  - a (BPEL) workflow integrates external Web services having given reliability and we wish to assess reliability of composition
  - if reliability of an external service changes, does our property about reliability of composition change?
    - ✓ our previous work framed this into probabilistic model checking
    - ✓ here we can deal with unrestricted changes, also in the workflow in a very efficient way

Initial validation of the approach

- Case 2: reachability analysis as supported by program model checking
  - given a program and a safety property, is there an execution of the program that leads to a violation of the property?
  - if the program changes, how does our property change?
    - ✓ similar problem faced by Henzinger et al.
    - T. A. Henzinger, R. Jhala, R. Majumdar, and M. A. Sanvido. Extreme model checking. In Verification: Theory and Practice, volume 2772 of LNCS, 2004.



Zoom-in

Control-theory based self adaptation

[ASE 2011] A. Filieri, C. Ghezzi, A. Leva, M. Maggio,
Self-Adaptive Software Meets Control Theory: A Preliminary Approach Supporting Reliability

Goal

- Tune software through its model via feedback control loop
- Formally prove controller's capabilities (error reduction, convergence, ...)

Conclusions and future work

- (Self-)adaptation is needed
- It requires a paradigm shift
- Run-time environments must become semantically rich
- Run-time **reasoning** must be supported, not just execution
- Continuous change and the quest for incrementality
- Benefits can be achieved by applying methods from control theory

Thanks to the group

Acknowledgements

This work is supported by and Advanced Grant of the European Research Council

Programme IDEAS-ERC, Project 227977---SMScom