Software verification for ubiquitous computing

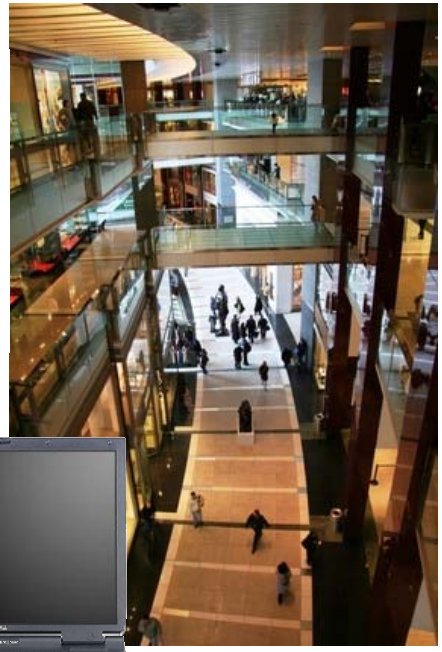
Marta Kwiatkowska

Computing Laboratory, University of Oxford

QA'09, Grenoble, June 2009

Software everywhere

- Electronic devices, ever smaller
  - Laptops, phones, sensors…
- Networking
  - Wireless & Internet everywhere
- Intelligent spaces
  - Buildings, vehicles…
- Systems
  - Self-*
  - Adaptive
  - Context-aware



- From hardware and software, to everyware
  - Household objects do information processing
  - Software is central

Challenges

- Communication: increased rate of failure
  - Wireless medium
  - Low power, mobility
- Coordination: increased use of randomisation
  - Distributed computation, de-centralisation
  - Mobility, highly dynamic behaviour
- Context: increased intelligence to ensure smart adaptation
  - Sensors are integral components of devices
- and more…

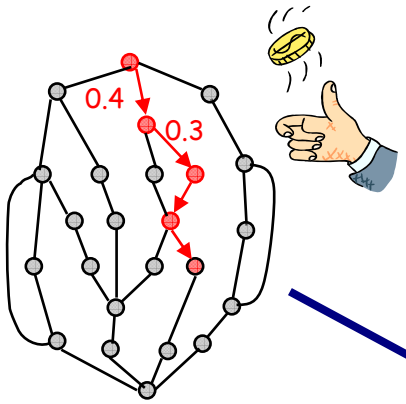
- How to ensure correctness, safety, dependability, security, performability?
  - Complex scenarios, recovery from faults, resource usage, …

Formal verification to the rescue

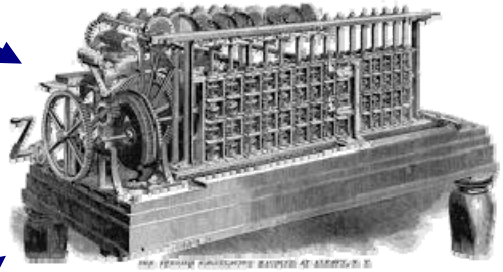
- Prevailing paradigm: Model, Analyse, Improve Design
  - Is it appropriate?
- Must include quantitative, as well as qualitative, requirements:
  - For battery, "expected power consumption is …"
  - For denial of service, "expected cost of response is …"
  - For communication protocols with transmission failures, "maximum probability of delivery within time deadline is …"
- Models are typically extensions of automata annotated with quantitative information, e.g.
  - Probability of taking a transition
  - Time deadline
  - Amount of resource

Quantitative probabilistic model checking…

in a nutshell



Probabilistic model, with resources



Probabilistic Model Checker

or

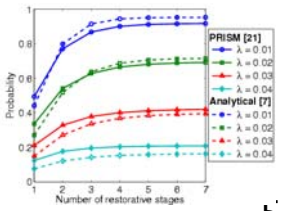
or

Probability/Expectation

send → $P_{\geq p}$ [F deliver]

Probabilistic temporal logic specification

State 5: 0.6789
State 6: 0.9789
State 7: 1.0
…
State 12: 0
State 13: 0.1245



Why quantitative verification?

- Analysis combines "quantitative" and "exhaustive" aspects
- Best/worst-case scenarios, not possible with simulation
  - Computing values for a range of states
    - "maximum expected run-time over all possible initial configurations"
  - All possible resolutions of nondeterminism
    - "maximum expected number of bits revealed under any eavesdropping strategy"
- Identifying trends and anomalies
  - Counterexamples (error traces)
  - Experiments: ranges of model/property parameters
    - e.g. N=1..5, T=1..100, for N model parameter and T a time bound
    - identify patterns, trends, anomalies in quantitative results

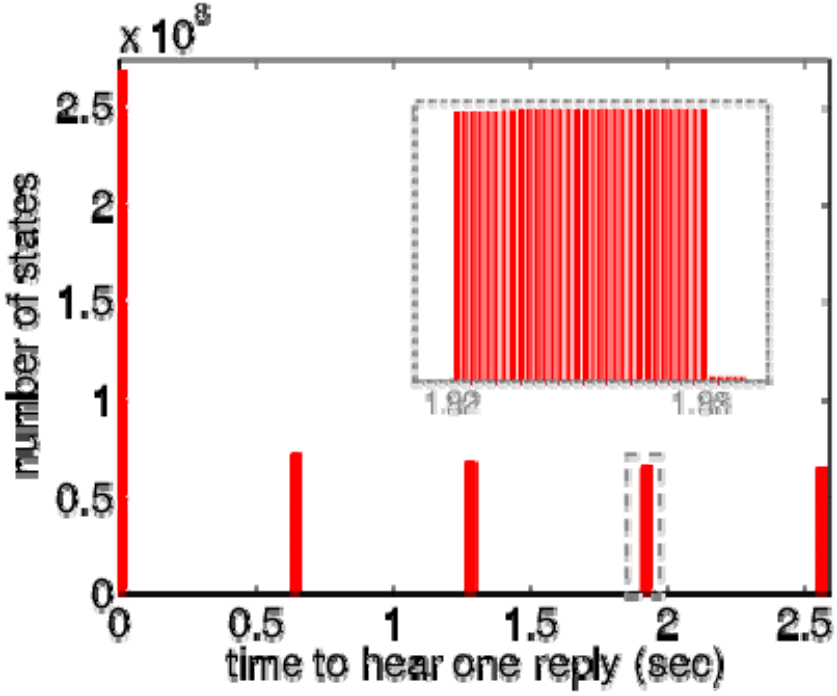
Some achievements – Bluetooth

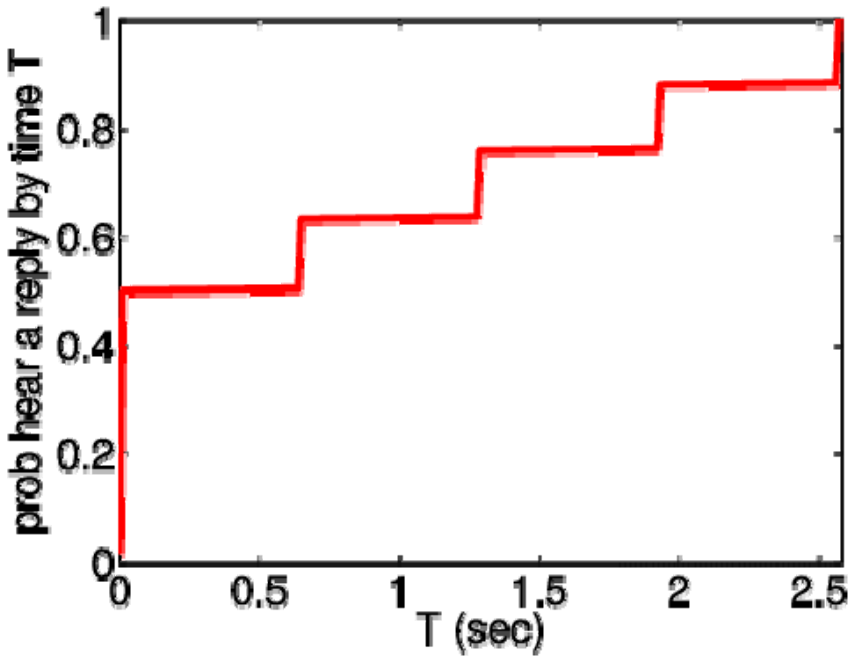
- Bluetooth device discovery – Huge model!
  - complex interaction between sender/receiver
  - genuine randomness – discrete time Markov chain model
  - sender/receiver not initially synchronised, huge number of possible initial configurations (17,179,869,184)
  - initially, model checking infeasible
  - partition into 32 scenarios, i.e. 32 separate DTMCs
  - on average, approx. 3.4×10^9 states, 536,870,912 initial

- Property model checked:
  - "worst-case (maximum) expected time to hear K replies, over all possible initial configurations"
  - also: how many initial states for each possible expected time
  - and: cumulative distribution function assuming equal probability for each initial state

Bluetooth – Time to hear 1 reply

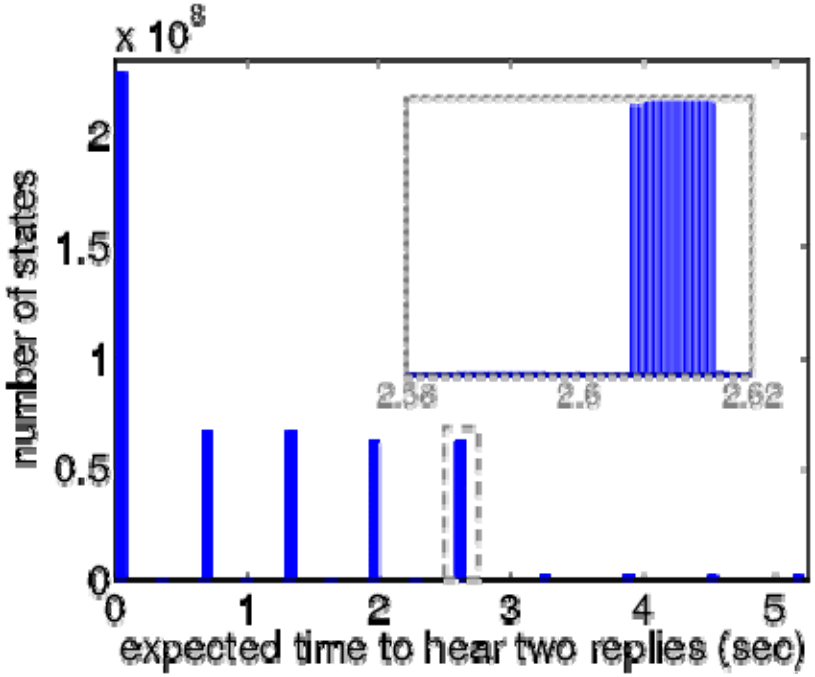
- Worst-case expected time = 2.5716s
  - in 921,600 possible initial states
- Best-case expected time = 635μs

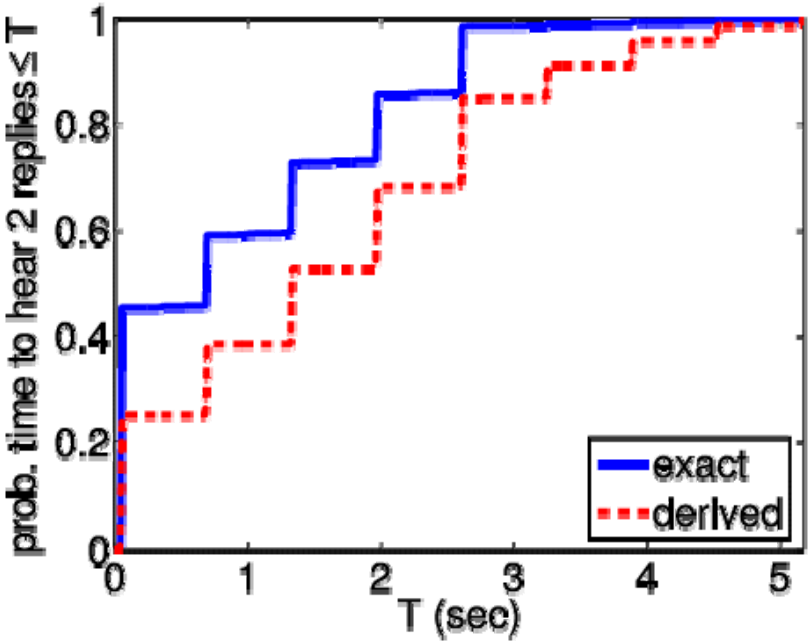




Bluetooth – Time to hear 2 replies

- Worst-case expected time = 5.177s
  - in 444 possible initial states
- Compare actual CDF with derived version which assumes times to reply to first/second messages are independent



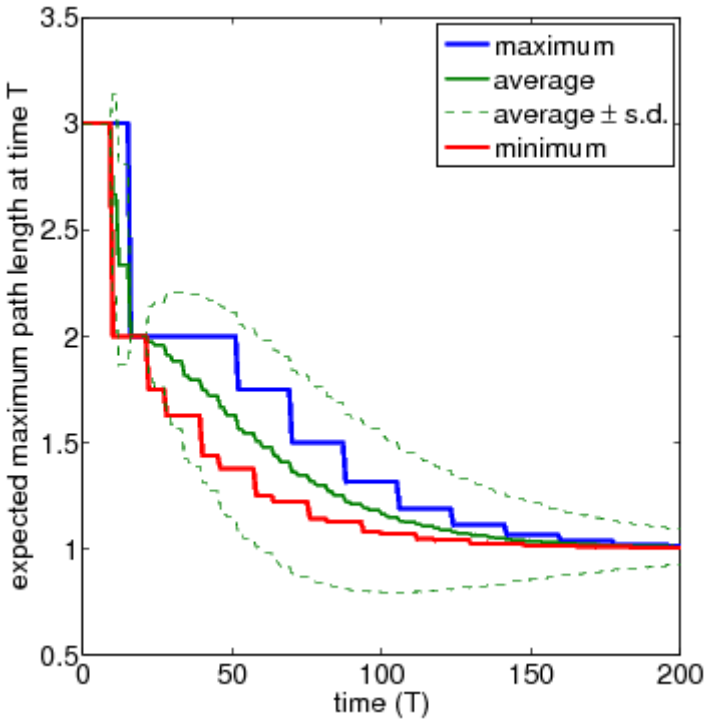


Some achievements – Gossip

- Gossip protocols: work by spreading rumours
  - N nodes, each with address
  - nodes maintain a partial view of network topology: list of up to c (<N) node descriptors (addresses and freshness)
  - nodes periodically exchange information with a number of its peers, keep c newest views
  - Markov decision process model
    - random peer selection
    - freshness based on hop-counts

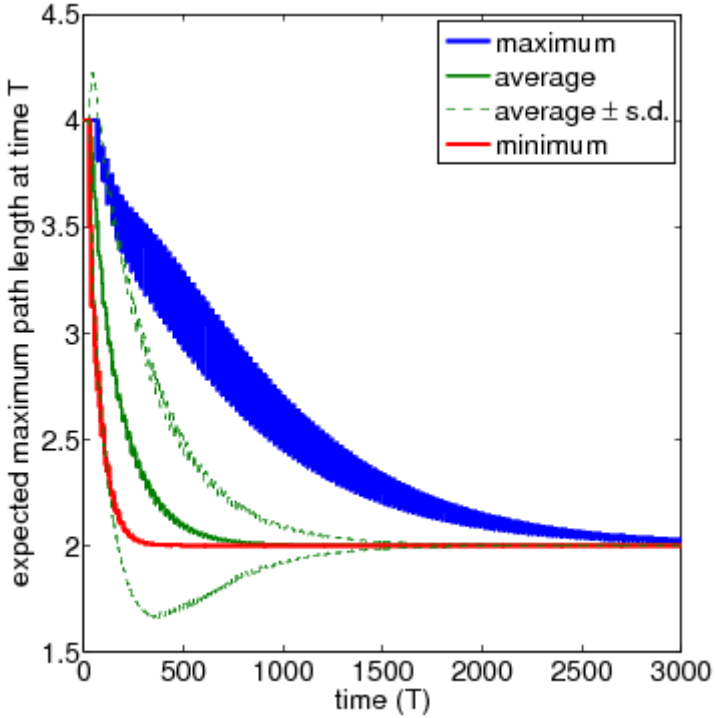
- Analyse performance and how network topology varies over time through partial views
  - (minimum and maximum) expected path length (longest route between nodes)
  - (minimum and maximum) expected number of gossiping rounds required to reach connectivity

Gossip – Results



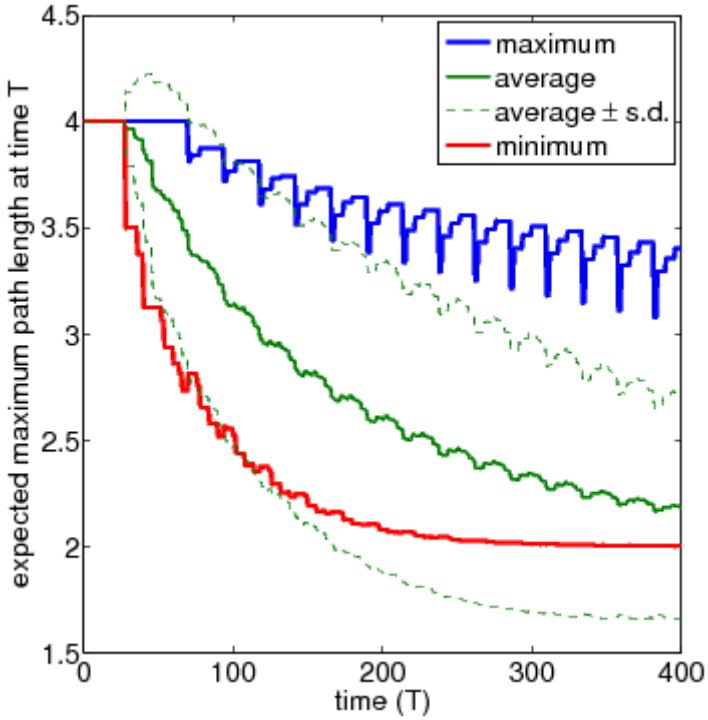
The expected path length for N=3

Gossip – Results



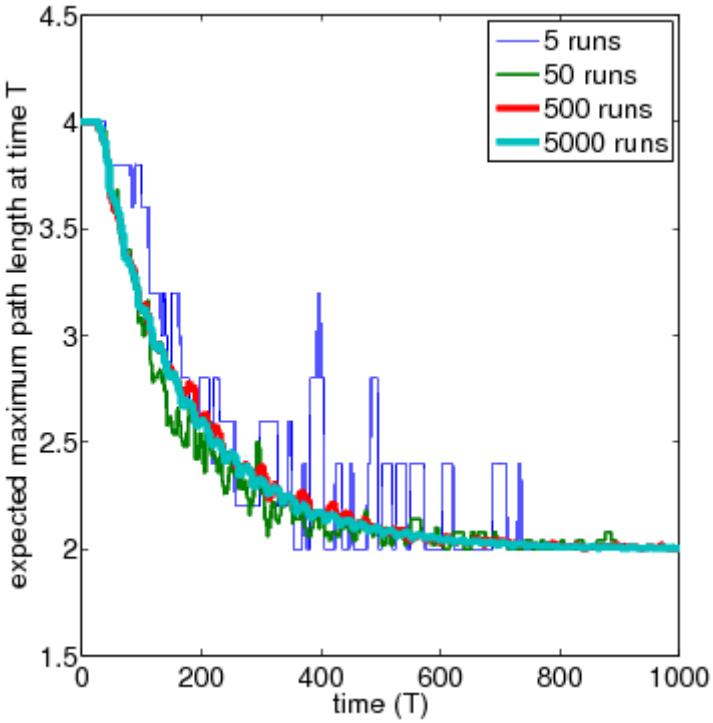
The expected path length for N=4

Gossip – Results



The expected path length for N=4 (zoomed in)

Gossip – Results



Average simulation results for N=4 (zoomed in)

What have we learnt?

- Realistic ubiquitous computing scenarios!
- Bluetooth device discovery [DKNP06]
  - State space explosion
    - only 2 devices (sender and receiver), up to K=2 messages
  - Independence assumptions can be misleading
  - Nevertheless, able to calculate detailed best/worst case time
- Gossip protocols [KNP08]
  - State space explosion (again), only 3 and 4 nodes
  - Complexity of the scenario
    - Great variability of expected path length
    - Simulation results (averaged) often not indicative of true performance characteristics
- See www.prismmodelchecker.org for more information

But can we deal with everyware?

- Current state of affairs
  - Models: mostly manually derived, not real software
  - Agents: no self, cooperation/negotiation little studied
  - Requirements: pre-hoc, violated under system adaptation
  - Quantitative verification: offline, largely monolithic
- Some possible directions
  - Software verification for randomisation/failures
  - Quantitative run-time verification for autonomous systems
  - and more…
    - Software verification for context-awareness
    - Synthesis from quantitative objectives
    - Game-based quantitative approaches
    - Quantitative component-based design

Software verification for randomisation

- Combine software verification via model checking
  - efficient verification of C/Java via abstraction-refinement
  - e.g. "buffer overflow does not occur in this program, for all executions"
  - existing tools: SLAM, BLAST, SATABS, Terminator
- with quantitative probabilistic model checking
  - focus on software where failure is unavoidable
    - e.g. network protocols, esp. wireless
  - aim for network utilities and quantitative properties
    - concurrency, user input -> nondeterministic choice
    - failure, randomisation -> probabilistic choice
  - also include real-time and resources

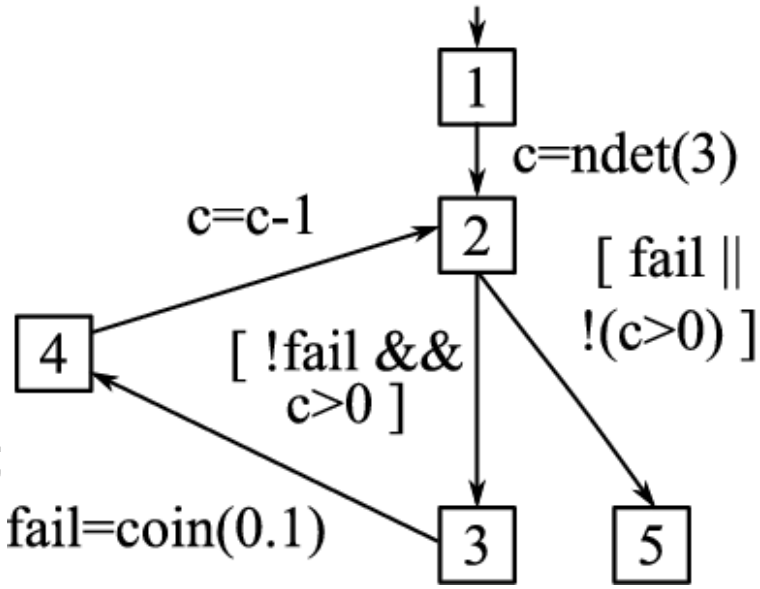
Probabilistic software

- Consider ANSI C programs
  - support functions, pointers, arrays, but not dynamic memory allocation, unbounded recursion, floating point ops
  - no concurrency, yet
- Add function bool coin(double p) for probabilistic choice
  - for modelling e.g. failures, randomisation
- Add function int ndet(int n) for nondeterministic choice
  - for modelling e.g. user input, unspecified function calls
- Focus on software where failure is unavoidable
  - e.g. network protocols/utilities, esp. wireless
- Quantitative properties based on probabilistic reachability
  - e.g. minimum expected number of packets sent

Probabilistic program example

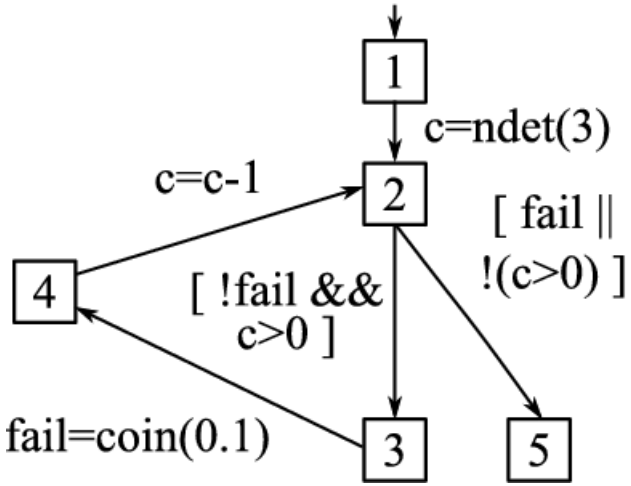
```
bool fail = false;
int c = 0;
int main ()
{
        // nondeterministic
        c = ndet (3);
        while (! fail && c > 0)
        {
                // probabilistic
                fail = coin (0.1);
                c --;
        }
}
```

Probabilistic program

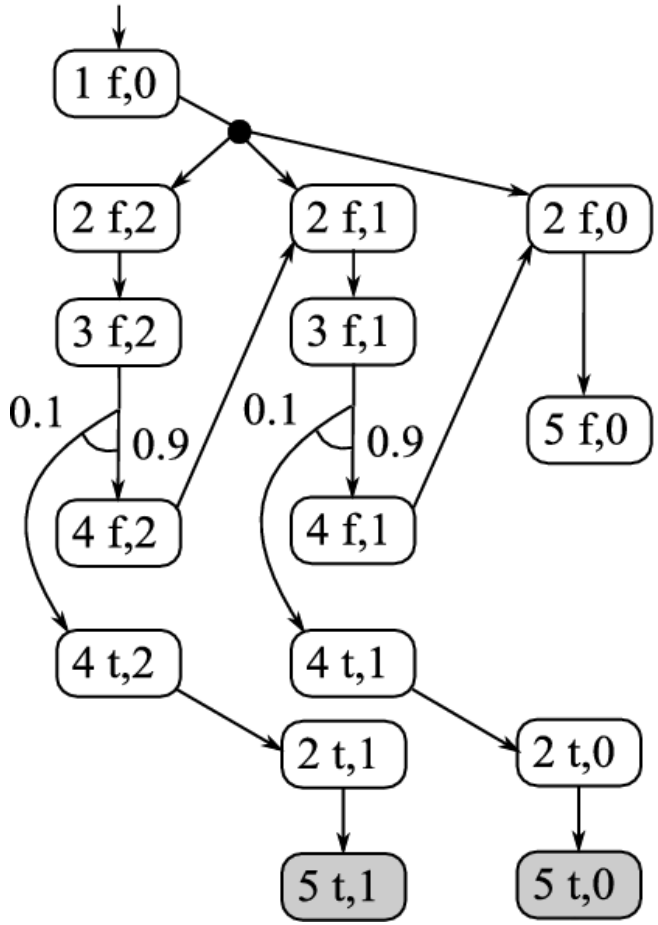


Probabilistic program as MDP

Probabilistic program



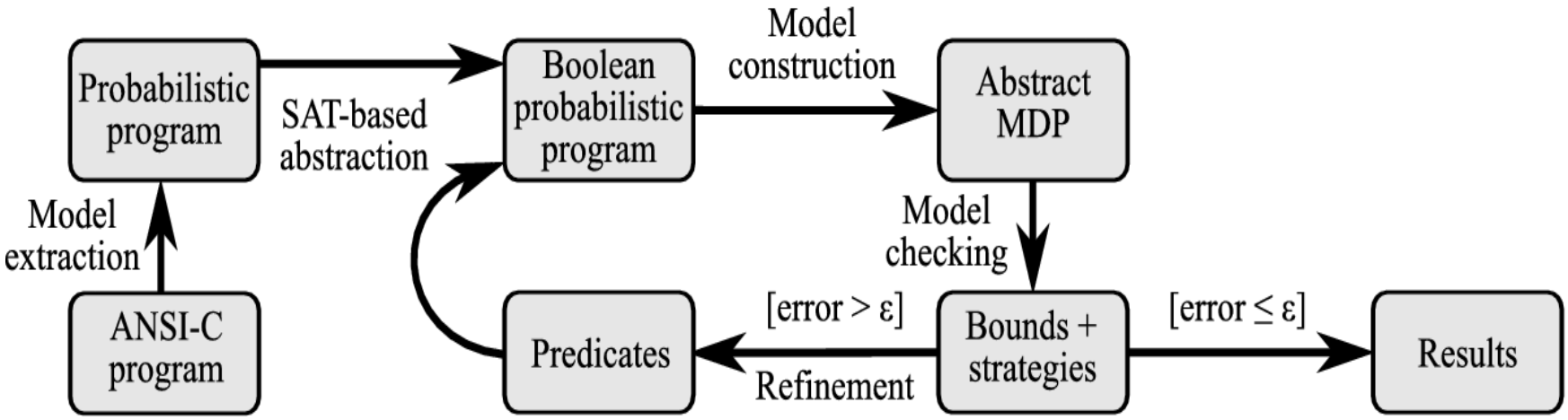
MDP semantics



minimum/maximum probability of the program terminating with fail being true is 0 and 0.19, respectively

How it works – the tool chain

- Model extraction: goto-cc
- SAT-based abstraction: SATABS
- Model checking: PRISM
- Abstraction/refinement: stochastic games [QEST06]
- For more information, see [VMCAI09]



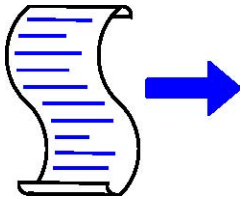
Experimental results

- Successfully applied to several Linux network utilities:
  - PING (tool for establishing network connectivity)
  - TFTP (file-transfer protocol client)
- Code characteristics
  - 1 KLOC of non-trivial ANSI-C code
  - Loss of packets modelled by probabilistic choice
  - Linux kernel calls modelled by nondeterministic choice
- Example properties
  - “maximum probability of establishing a write request”
  - “maximum expected amount of data that is sent before timeout”
  - “maximum expected number of echo requests required to establish connectivity”

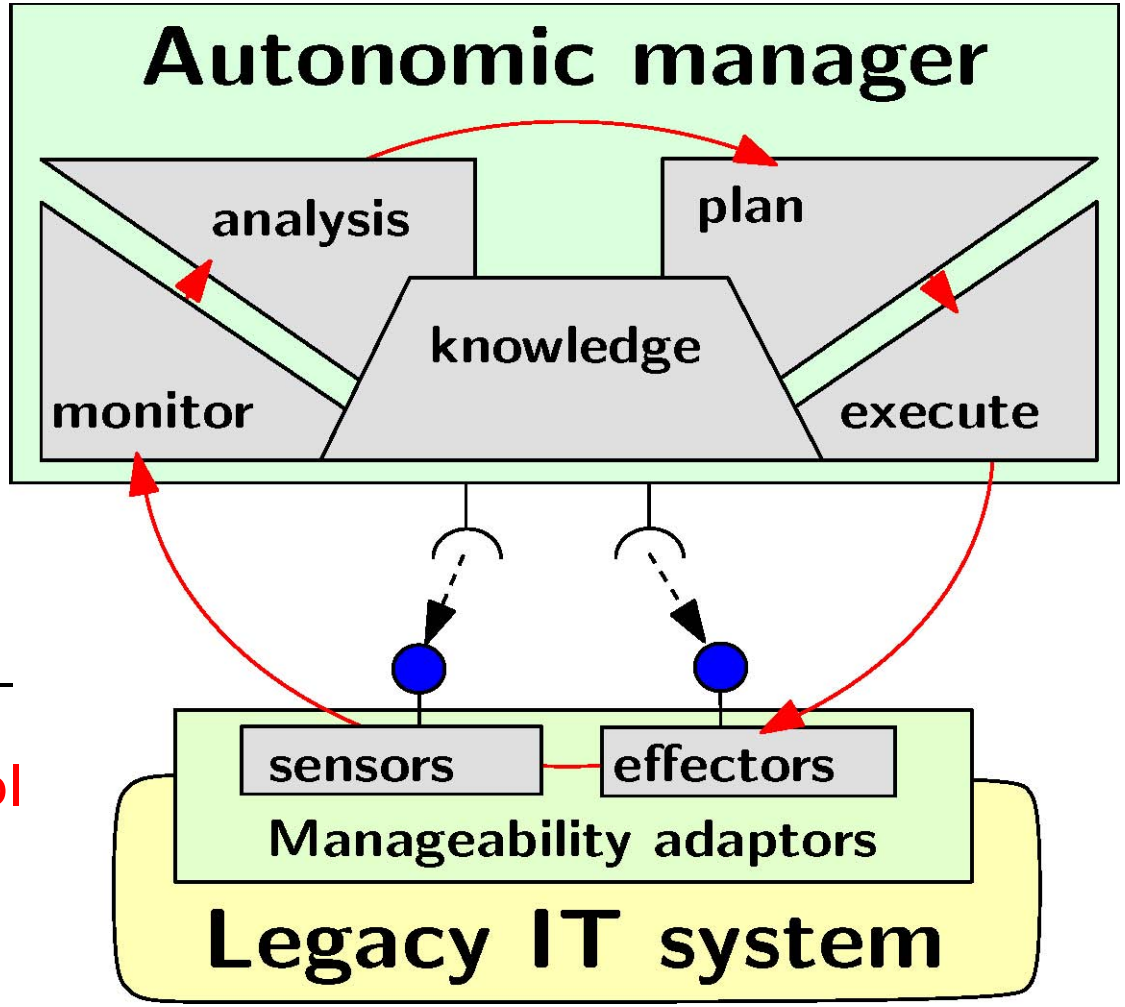
Quantitative run-time verification

- Autonomic systems (ubicomp, legacy systems, etc)
  - Require increasingly demanding non-functional requirements
    - performance, dependability, utility, …
  - Need to adapt to changing scenarios
    - workload, environment, objectives, …
- Key observations: offline qualitative verification methods do not suffice
- Instead integrate [ICSE09] [FASE09]:
  - Multi-objective quantitative run-time analysis
    - rigorous exhaustive checks
  - Self-* adaptive system capabilities
    - adaptation decisions based on (non-linear) behaviour

Background: autonomic systems

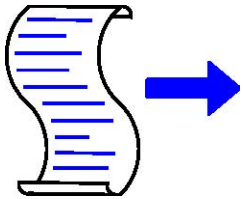


System objectives (policies)

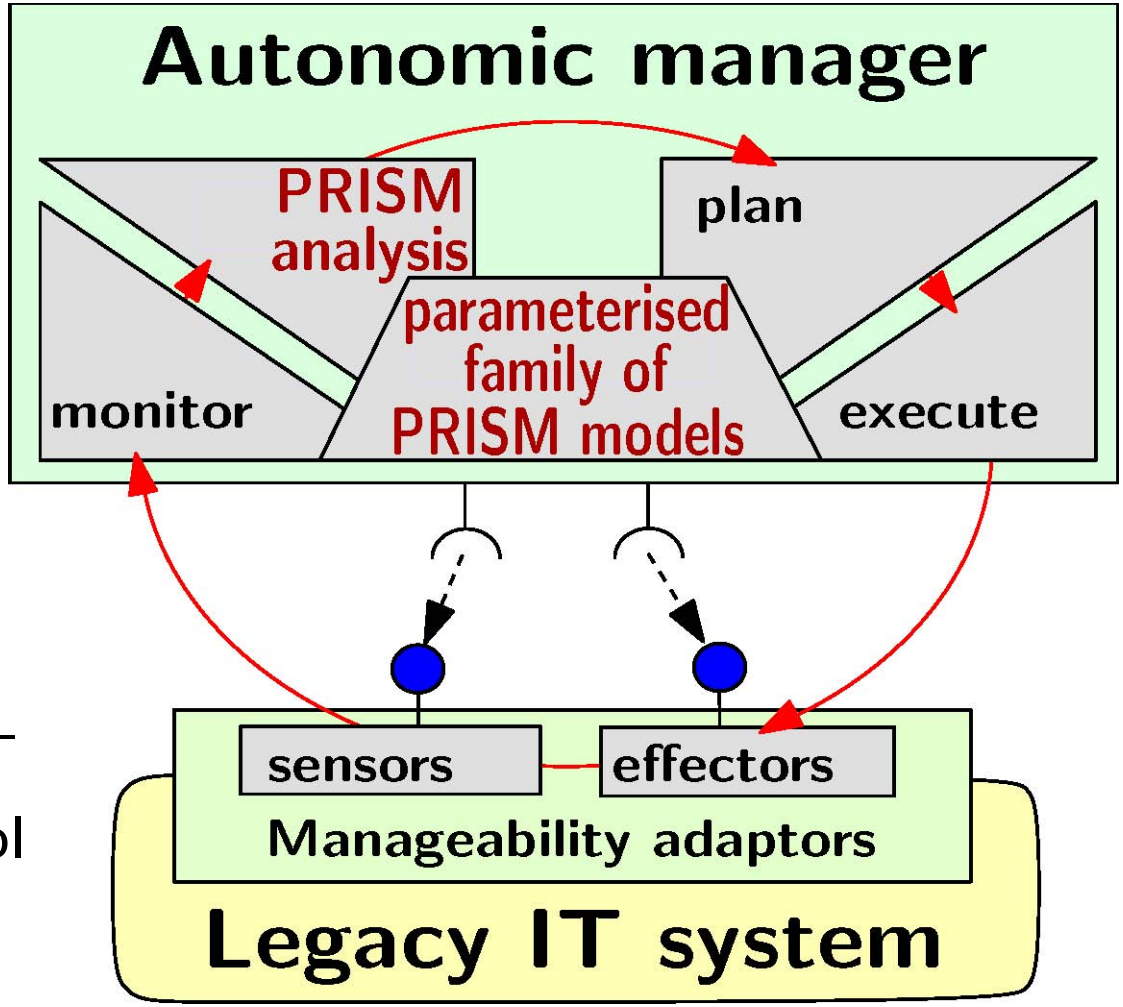


monitor–analyse–plan–execute autonomic control loop

Integration

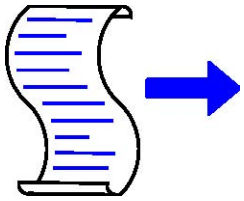


System objectives (policies)

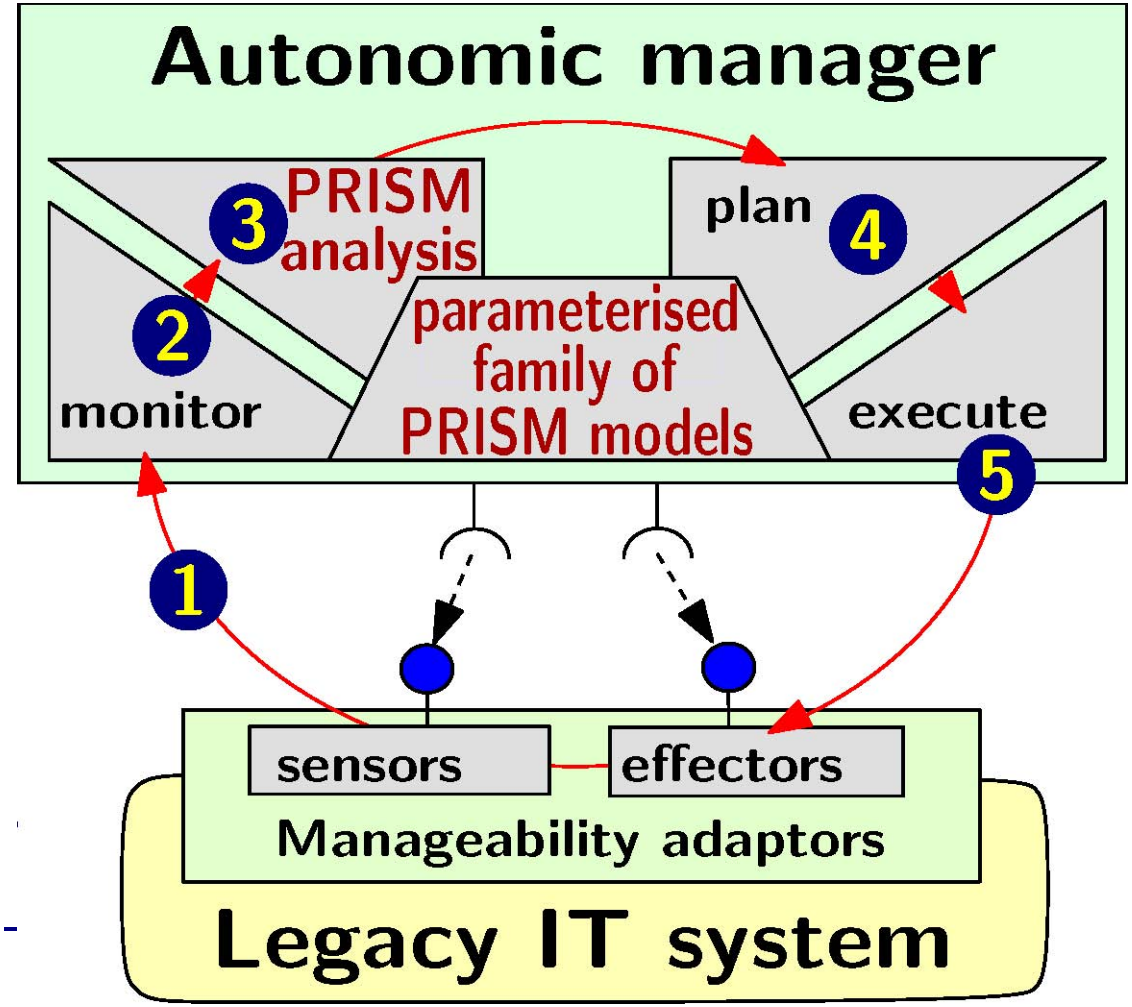


PRISM-driven monitor-analyse-plan-execute autonomic control loop

Integration



System objectives (policies)



1. Monitor
2. Select subset of models and configurations
3. Carry out PRISM experiment
4. Choose best configuration
5. Enforce chosen configuration

How it works

- Development method
  - Assume exposed control parameters
  - Model-driven generation of adaptor code
  - System objectives specified by administrator/designer
    - E.g. minimise power consumption and maximise duration
  - At run-time, monitor via quantitative verification and influence the system by modifying control parameters

- Case study: adaptive power-management
  - Adjust configuration automatically
    - to satisfy objectives (energy, response time), and
    - to reflect workload changes (inter-arrival rate)
  - Simulation based, applicable to disk drives (Fujitsu data), etc

Conclusions and further work

- Quantitative verification for adaptive systems
  - combine with statistical inference
  - develop incremental verification methods
- Software verification for sensor networks
  - working with NesC/TinyOS with threads
  - focus on software errors due to interface misuse, context changes and/or concurrency
  - adaptation of goto-cc and SATABS
- What about everyware? We are only just beginning to realise the true complexity of ubicomp systems
  - stochastic behaviours, continuous/discrete evolution, cooperation/negotiation and self-*, etc