# Testing software systems – a perspective

**Gregor v. Bochmann**

School of Electrical Engineering and Computer Science

University of Ottawa, Canada

ICTSS 2015

**Sharjah and Dubai (UAE), November 2015**

# Abstract


The talk will begin with a review of general testing concepts, such as white-box and black-box testing, different realizations of oracles (including a formal behavior specification), fault models and fault coverage issues, and testing architectures. This will set the framework for the following discussion which has two parts: (a) a discussion of the history of the ICTSS conference and the issues discussed during the early times since around 1985, and (b) an overview of two ongoing research projects: (1) on testing implementations against partial-order specifications, and (2) on reverse engineering of Rich Internet Applications for vulnerability testing.

The first ICTSS conference was held in Vancouver (Canada) in 1988 and was called International Workshop on Protocol Test Systems. The main question discussed at that time was how to test a protocol implementation to ensure that it satisfies all requirements of a given protocol specification (a form of black-box testing). The main issues were the modeling language used for the specification, fault models, and algorithms for obtaining test suites with given fault coverage. At the same time, standardization committees of ISO and ITU developed guidelines for architectures for protocol testing and a language (TTCN) for specifying test cases. Later, the scope of ICTSS was broadened to cover the testing of many other kinds of software systems.

In the second part of the talk, we will first discuss issues that arise in testing systems against a behavior specification that defines a partial order for the interactions of the implementation. Different partial-order specification languages will be considered. Then another ongoing research project on crawling Rich Internet Applications (RIAs) is discussed. Through the testing of a given implementation, a model of the RIA is developed (this is a kind of black-box testing, but without a reference specification). The purpose here is to obtain a "complete" model of the application such that each state (i.e. each page at the user interface) of the application can be subsequently checked for security vulnerabilities or accessibility requirements. Since the state space of these applications is usually huge, we propose (a) different algorithms for obtaining the most important information relatively fast, (b) concurrent exploration by multiple crawlers, and (c) some methods for avoiding the exploration of "equivalent" and "redundant" states.

# Which topics for this talk ?

- I was much involved in research on protocol testing in the 1980ies and '90ies
- But since 2000 mainly working in other fields
- Here is a photo from IWPTS (International Workshop on Protocol Test Systems) in Pau (France) – 1993
  - This was **for me** one of the high times of this conference

# IWPTS 1993 - Photo

# Outline of talk

- Historical perspective
  - Model-based development
  - State machine testing
- An on-going project: Crawling Rich Internet Applications (RIA)
  - Testing in the software engineering process
  - A testing approach to retro-engineering of RIA in view of security testing
- Conclusions

# Part 1: Historical perspective

- **Milestones for distributed systems development**
  - First computer networks (around 1972)
  - First computer network **standards** (X.25 – 1976)
  - OSI and ODP standardization (approx. 1980 – 95)
    - Much interest in testing protocol implementations against standards
  - Commercial systems for protocol testing
    - E.g. Idacom – HP 's protocol tester for X.25, Frame Relay, ATM, etc.
  - Public use of the Internet (since around 1995)
  - Wireless communication standards, GSM, etc.

# Standardization group on OSI conformance testing

- Led by Dave Rayner (UK) from 1983 to 1997.
- Developed a comprehensive ISO and ITU standard on protocol conformance testing ("guidelines")
  - General concepts and possible architectures
  - TTCN language for specifying abstract test cases
  - Additional information required for testing
- This standard was later used for defining standardized test suites for other protocols, such as GSM, Internet, etc.

# My research areas

- At the Université de Montréal
  - 1972 – neural networks
  - 1973 – compilation and semantic attributes
  - Since 1975 – protocol specification, verification
  - Early '80ies – standardization of FDT's
    - Three FDT's were developed: Estelle, SDL and LOTOS
    - Rayner's group did not endorse any, but developed TTCN
  - Since 1982 – protocol testing
    - 1989 – 1997 : Industrial research chair with IDACOM-HP
- At the University of Ottawa - also other topics:
  - QoS at the application level - P2P systems - optical networks - crawling RIA's
- Recurring themes: submodule derivation (since 1980) and protocol derivation (since 1986)

# International conferences on protocol engineering

- Protocol Specification, Testing and Verification (PSTV)
  - 1981 – first PSTV
  - 1988 – first FORTE (Formal Description Techniques)
  - 1996 – PSTV-FORTE combined
  - 2009 – combined with FMOODS (Formal Methods for Open Object-Based Distributed Systems) – now called FORTE : “Formal Techniques for Distributed Objects, Components and Systems”
- ICTSS
  - 1988 – first IWPTS (International Workshop on Protocol Test Systems)
  - 1997 – called International Workshop on Testing Communicating Systems
  - 2000 – called TestCom
  - 2007 – combined with FATES (Formal Approaches to Software Testing, founded 2001)
  - 2010 – called ICTSS (this is a more general theme, not only distributed systems)

# Part 2: Model-based development

- **Model-based development**
  - This is an expression much used with design or requirements models given in UML (which was defined around 1995)
  - Model-based development was actively pursued since the mid-1970ies for the development of communication protocols
  - Since the behavior of protocol entities can be largely described by state machines, the models used were often state machine models.

# **Testing methodology:** There are always two issues:

- **Test coverage :**
  - It is impossible to test the IUT for all possible behavior sequences.
  - How can one select a (not too big) set of test cases that would discover as many faults as possible among the faults that are expected to be present in the IUT ? – *This implies two questions:*
    - What are the expected faults (also called **fault model**) ?
    - What set of test cases would be most effective ?
- **Test result evaluation:**
  - After a test case has been applied to the IUT and the outputs of the IUT have been observed, how does one determine whether the observed behavior is conform to the specification ?

# Traditional software testing methodology

- **White-box testing :** tests developed from knowledge of the program being tested
  - **Test coverage:**
    - There is no clear fault model.
      - **Mutation testing** is sometimes used to determine the fault coverage of a given test suite. The mutations introduced represent the fault model.
    - To define test coverage, one uses **test coverage criteria**
      - **Criteria based on program structure:**
        - **All branches**
        - **All paths**
        - **Data-flow criteria, such as all Def-Use pairs**
      - **Criteria based on input parameter variations:**
        - **Extreme and intermediate values (this is partly related to the structural criteria above)**

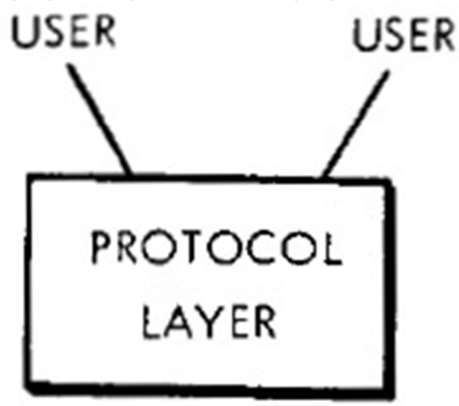
# Traditional software testing methodology

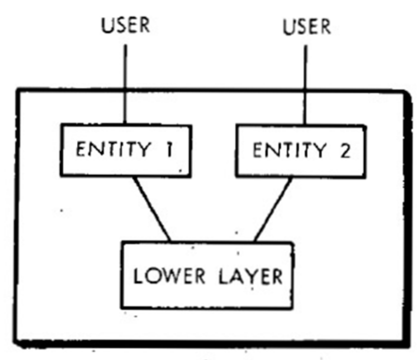
- **White-box testing :** tests developed from knowledge of the program being tested
  - **Test result evaluation:**
    - One often talks about the “**Oracle”** that analyses the output and determines whether a fault was detected
    - The word “oracle” suggests that there is no precise definition of the requirements on which such a decision could be based.
      - Often, the requirements are described quite informally
    - Usually, the test developer includes in the test program the analysis of the IUT output (based on his understanding of the requirements)

# Model-based development of protocols

- **Protocol specification:** a precise definition is required to assure compatibility between different protocol implementations. It is an abstract model of all implementations.
- **Service specification:** defines the abstract interactions of a protocol entity with the user, and the global properties to be assured by the communicating protocol entities.

Architectural views of service and protocol entities (from [], 1980)

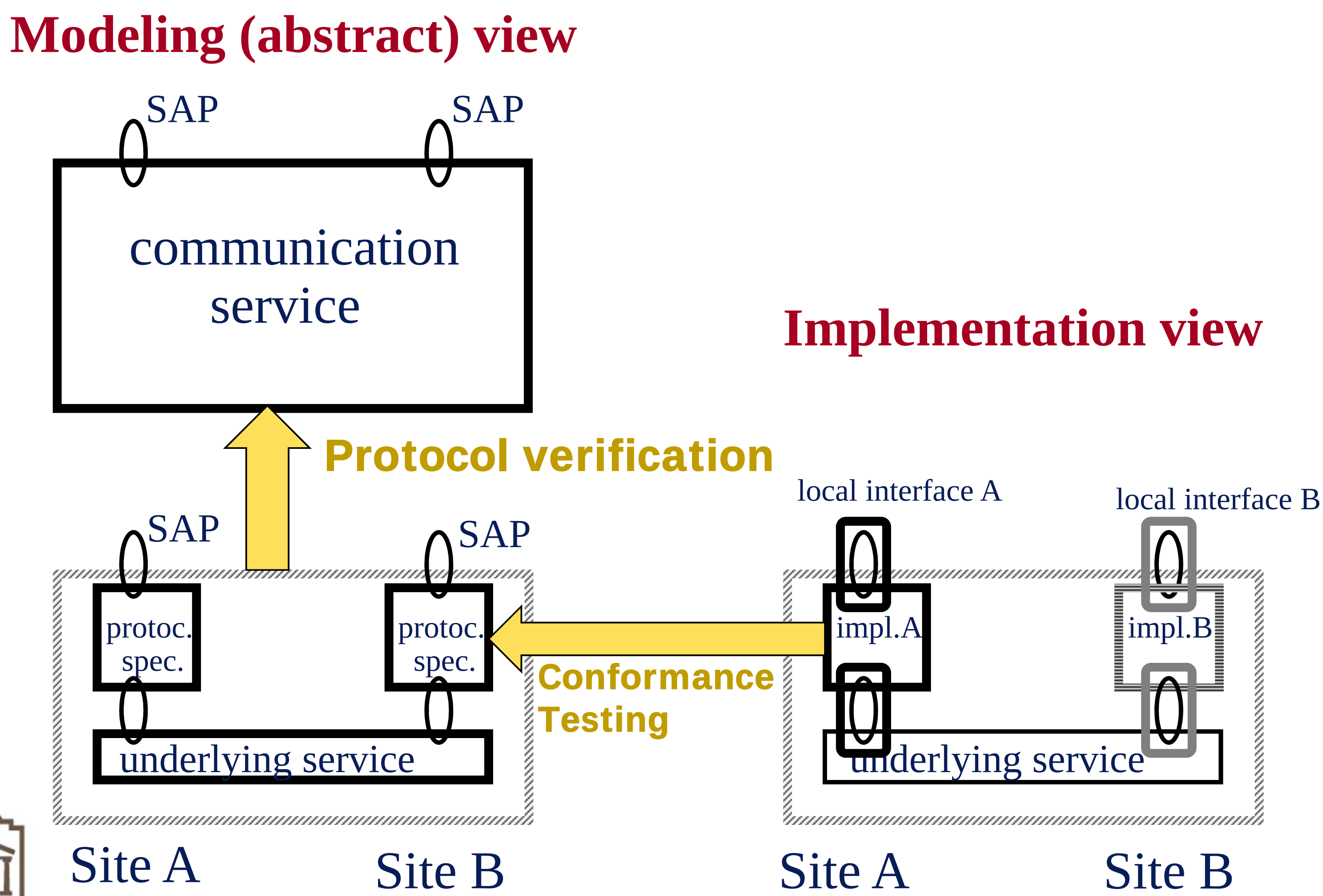


# V&V in protocol engineering

- **Protocol verification:** check that the protocol specification (the model) implies the service specification (a more abstract model).
  - This can be done by **model checking** or by **testing the protocol specification** (if the latter is executable)
- **Conformance testing:** check that a given implementation conforms to the protocol specification. --- Usually, one wants a test suite that can be applied to any implementation of the protocol
  - Therefore the test suite should be based on the protocol specification (the model), not the implementation code
  - This is **black-box testing** – nowadays often called **model-based testing**

# V&V in protocol engineering : Architectural views



# How is protocol testing different ?

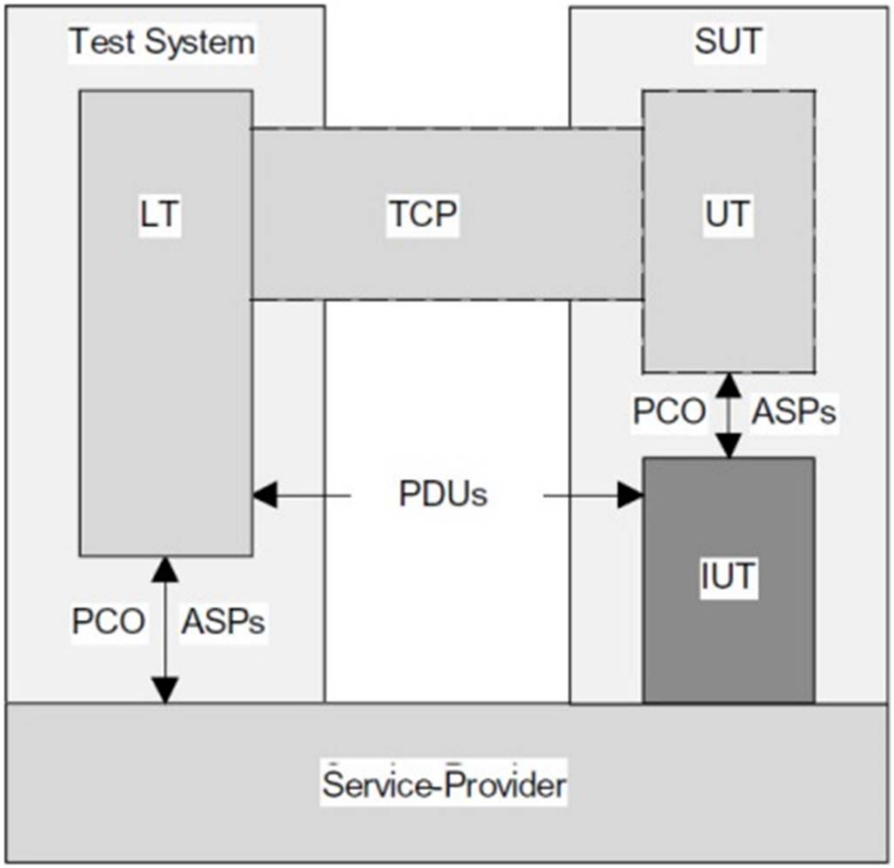
## There is a precise protocol specification

- and important aspects can be described by a **state machine model**

- **Test coverage :**
  - The state machine model suggests a precise **fault model**:
    - **Output faults** and **transfer faults**
  - Test coverage can be evaluated based on the fault model.
    - Some test suite development methods ensure "**full"** fault coverage
- **Test result evaluation :**
  - The protocol specification serves as oracle.
- **Observability and control issue**
  - The IUT has several interfaces

# Observability and control issue

**OSI Conformance Testing Methodology and Framework – General Concepts (X.290)**




**b) The Distributed test methods**

**Upper tester (UT) and Lower tester (LT)**

**A synchronizable test sequence can be executed without any test coordination protocol (TCP) between upper and lower tester**

**Different testing architectures**

- Local
- Distributed
- Coordinated
- Remote

test method

# Part 3: State machine testing

- Early 1980ies: First work on test suite design for protocol testing based on state machine models (with my PhD student Behcet Sarikaya)
- We found 3 existing test design methods using state machine models:
  - Distinguishing sequence (not feasible for all state machines)
  - Transition tour – similar to All-Branches criteria (incomplete coverage in case of transfer faults)
  - W-method - has **full fault coverage guarantee** under the assumption that number of states of IUT is not larger than spec.
- Sarikaya's contributions (journal paper 1984):
  - Development of test suites based on protocol specifications
  - Dealing with synchronization issues due to multiple interfaces
  - Slicing of extended state machine models based on data flow

# Characterizing the W-method

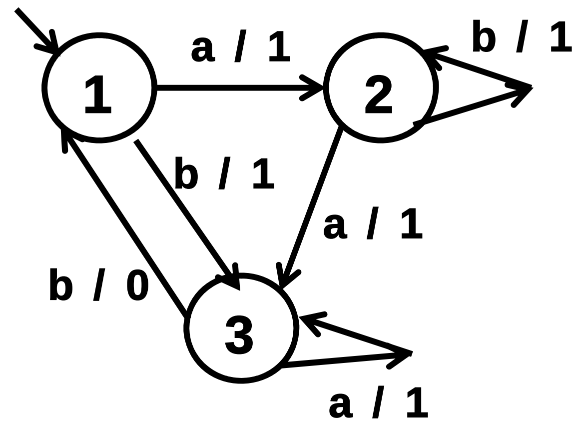
- A test suite developed by the W-method has two phases:

1. State identification: all states of the specification are identified in the IUT by leading the IUT into each state (possibly several times) and applying a set W of identification sequences to check that this state of the IUT shows the behavior foreseen by the specification.
2. Transition checking: Each transition is checked by executing it (possibly several times), observing the output and applying the W-set of sequences to check that the transition transfers to the right state.

- Assumption: The ITU has a reliable reset. Each test case starts with a reset and finishes with the execution of one of the sequences in the W-set.

# Simple example for the W-method

- Inputs = {a, b}
- Outputs = {0,1}
- W = {<a b>, <b>}
  - <b> distinguishes between state 3 and (1 or 2)
  - <a, b> distinguishes between state 1 and 2




**Output obtained from different states:**

| input | <a, b> | <b> |
|---|---|---|
| State 1: | <1, 1> | <1> |
| State 2: | <1, 0> | <1> |
| State 3: | <1, 0> | <0> |

**Test suite contains these sequences:**

*Identify initial state:* <r, a, b>, <r, b>,
*Identify state 2:* <r, a, a, b>, <r, a, b>,
*Identify state 3:* <r, a, a, a, b>, <r, a, a, b>
*Check transition b from state 1:* <r, b, a, b>, <r, b, b>
*etc. ...*

**Note: this machine has also a Distinguishing sequence: <b, b>**

# Improving the W-method

The W-method has been improved by several authors with the objective of obtaining shorter test suites.

- Wp method: use separate identification sets for each state of the specification
- UIO method (unique I-O) : applicable if the specification admits a single (unique) identification sequence for each state
- HIS method (harmonized identification sequences): designed for partially defined state machines - there is a sequence for distinguishing each pair of states

# Dealing with non-determinism

## A: Trace semantics

**A-1: Observably non-deterministic specification** (state is determined by observed sequence of inputs and outputs)

- Need for adaptive testing (next input may depend on previous outputs received)
- Question: Should IUT realize all non-deterministic choices ?
- In case of a non-deterministic IUT, tests must be repeated to explore all possible choices of the IUT.

**A-2: State-nondeterminism in the specification** (it may be in different states after a given sequence of inputs and outputs)

- As above
- The oracle function becomes an algorithm with concurrent exploration or back-tracking.

# Dealing with non-determinism

**B: Failure semantics** (Here one assumes that possible blocking behavior must be tested as well as valid execution traces)

- Different conformance relations can be considered: testing equivalence, reduction of non-determinism, etc.
- Test suite development mostly without fault coverage guarantee
- Most work in this area has been done in relation with the LOTOS specification language.

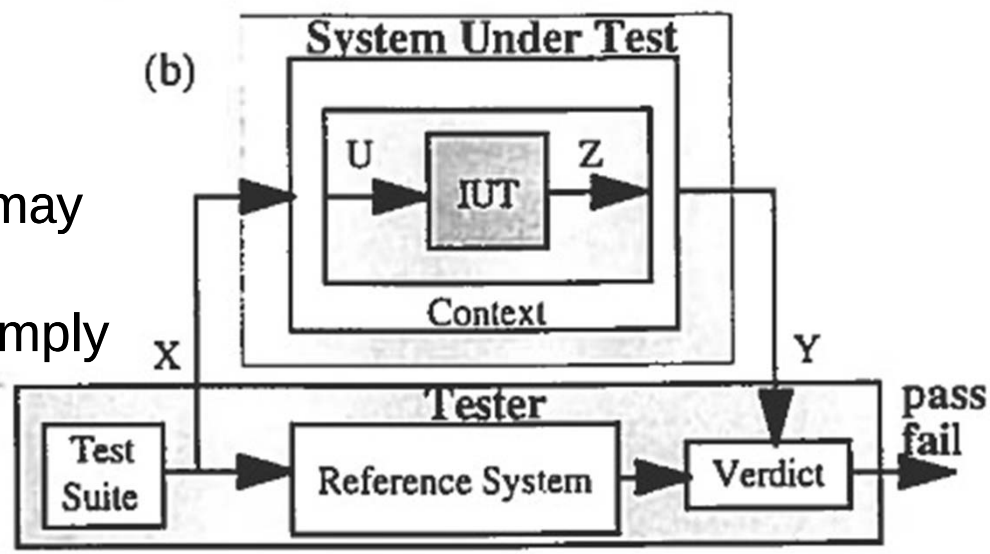
# Other issues

## Diagnostic testing

- Not only determine whether there is a fault in the IUT, but to locate the fault within the fault model
- Assumptions: (a) only output faults, (b) single fault, (c) multiple faults, but with restrictions

## Testing in context

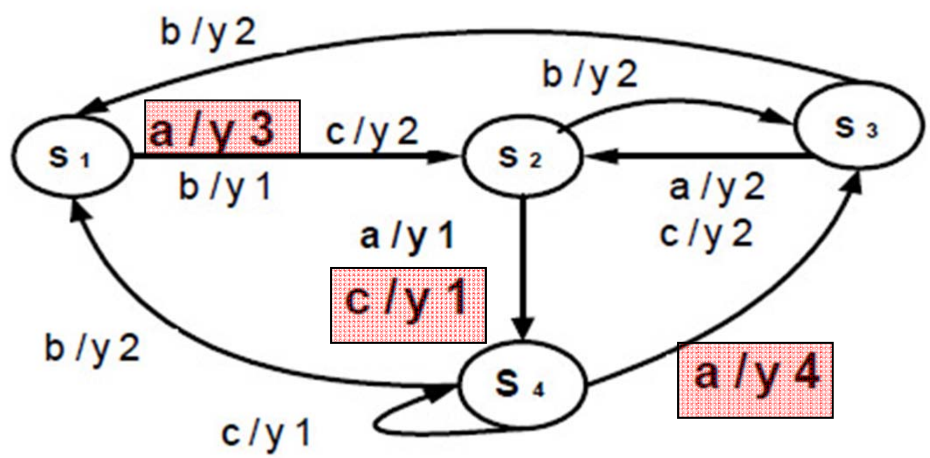
- IUT is embedded and its interfaces are not directly accessible – context behavior is known.
- Some deviations from the specified behavior of the IUT may not be detectable
- Which visible behavior would imply a fault in the IUT (reference system) ?
  - Submodule construction problem



# Other issues (ii)

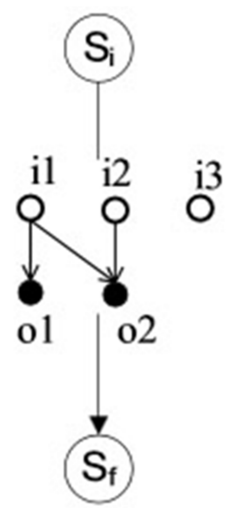
## Incremental testing

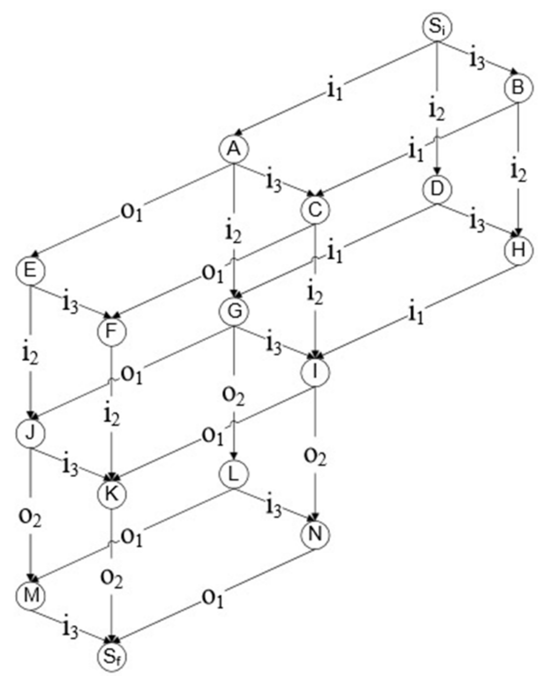
- Find identification sets without the modified transitions
- Test each modified transitions
- More complex with additional states




## Testing based on partial-order specifications

- Each transition has several inputs/outputs partially ordered
- Fault model based on the partial order
- Equivalent state machine would have much more states



# Questions concerning practical application

- **Q1:** Is it important to have a fault coverage guarantee (which is based on the assumption about the number of states of the IUT) ? – **One needs empirical evidence !!**
  - Is the assumption normally satisfied ?
  - What is the expected fault coverage when the assumption is not satisfied ?
  - What is the expected fault coverage for other test suites of similar length ?
  - Why not simply use a **readState** message which will identify the current state ? – *This single sequence of one input replaces the W-set.*
- **Q2:** Most test suites with fault coverage guarantee consist of a large number of test cases that start with *reset*. – In case that the assumption above is not satisfied, one could expect that test suites containing longer test cases (e.g. based on a Distinguishing sequence) would have a better chance of detecting certain faults due to additional states. – Is this true ? - **empirical evidence ??**

# Observations

- **O1:** State machine testing methods can be used for white-box testing:
  - If the IUT implementation has the structure of a state machine, a test suite can be derived from this state machine (e.g. using the W-method).
  - The output of the IUT could be checked by an oracle.
  - Under the assumption that the oracle is organised as a state machine with a number of states not larger than the IUT, the derived test suite will have full fault coverage.

# Observations

- **O2:** Test coverage criteria for black-box testing:
  - If the specification is written in some high-level programming/specification language, a test suite can be developed from this specification satisfying some given coverage criteria (like those developed for white-box testing of programs).
  - The specification could also serve as oracle.
  - There is no fault coverage guarantee, but mutation testing (mutating the specification) could be used to estimate the fault coverage of the test suite.
- **Note:** In general, model-based testing must be complemented with test cases that take the specific structure of the implementation into account (white-box).

# Testing extended state machines

- **Fact:** In most practical cases, a (simple) state machine model is only an approximation of the desired behavior of the IUT. Therefore one often uses **extended state machine models** for representing the behavior requirements.
  - These are state machines with additional state variables and input and output interactions that may contain parameters.
  - The behavioral aspects of the extensions are defined for each transition by:
    - An enabling predicate
    - An actions to be performed during the transition which determines the parameter values of the output interaction, and may update variables.

# Testing extended state machines (ii)

- The notation for defining these extensions is related to programming language concepts.
- Following the observation O2 above, it is therefore natural to use test coverage criteria (from software testing) for testing the behavioral aspects of the state machine extensions.
- This leads to combining state machine testing methods with data-flow test criteria (from software testing)
- Much work has been done in this area, but things are complex:
  - There are no fault coverage guarantees, and
  - Determining whether a given path is executable is undecidable

# Part 4: Testing in the software development process

## (A) Bug finding

- through testing (there is the coverage issue)
  - Implementation code is executed and tested
  - Design model is executed and tested
- through model checking
  - of the implementation code, or the design model
  - Coverage issue is solved by considering all execution paths – however, there may be state space explosion

# Part 4: Testing in the software development process

## (B) Reliability evaluation

- through testing with user input sequences that have the same probability distribution as in real operating conditions
  - These probability choices concern
    - Different choices of user inputs in each given state
    - Different choices of input data within the range of possibilities – with the same value distribution as in the real operating environment
  - One needs a probabilistic model of the user behavior
    - which can be obtained from observed user traces

# Part 4: Testing in the software development process

## (C) Other usages of testing

- Regression testing
- Test-driven (agile) software development
  - The requirements are given in the form of a test suite that includes the expected output
- Retro-engineering through testing
  - Application of tests to a black-box implementation for discovering its program structure
- Security testing
  - Apply specific security tests for exploring weaknesses in specific states of the application

# Part 4: Testing in the software development process

## (C) Other usages of testing

- . . .
- Retro-engineering through testing
  - Application of tests to a black-box implementation for discovering its program structure
- Security testing
  - Apply specific security tests for exploring weaknesses in specific states of the application
- **This leads us into the last part of my presentation**

# Part 5: Crawling Rich Internet Appl. (RIA)

- We extract a state machine model from a RIA by testing – identifying all reachable states (pages)
- This is a research collaboration between the University of Ottawa and IBM-Canada.
- IBM is interested in security testing

***Professors*** : Gregor v. Bochmann and Guy-Vincent Jourdan

***IBM collaborator:*** Dr. Iosif Viorel Onut

***Postdoc:*** Faheem Muhammad

***Students:***

| | |
|---|---|
| Khaled Ben Hafaiedh (PhD) | Sara Baghbanzadeh (M) |
| Salman Hoosmand (PhD) | Akib Mahmud (M) |

***Alumni:***

| | | | |
|---|---|---|---|
| Seyed M. Mir Taheri | (PhD) | Zou Di | (M) |
| Emre Dincturk | (PhD) | Suryakant Choudhary | (M) |
| Kamara Benjamin | (M) | Ali Moosavi | (M) |
| Xu Xinghao | (M) | | |

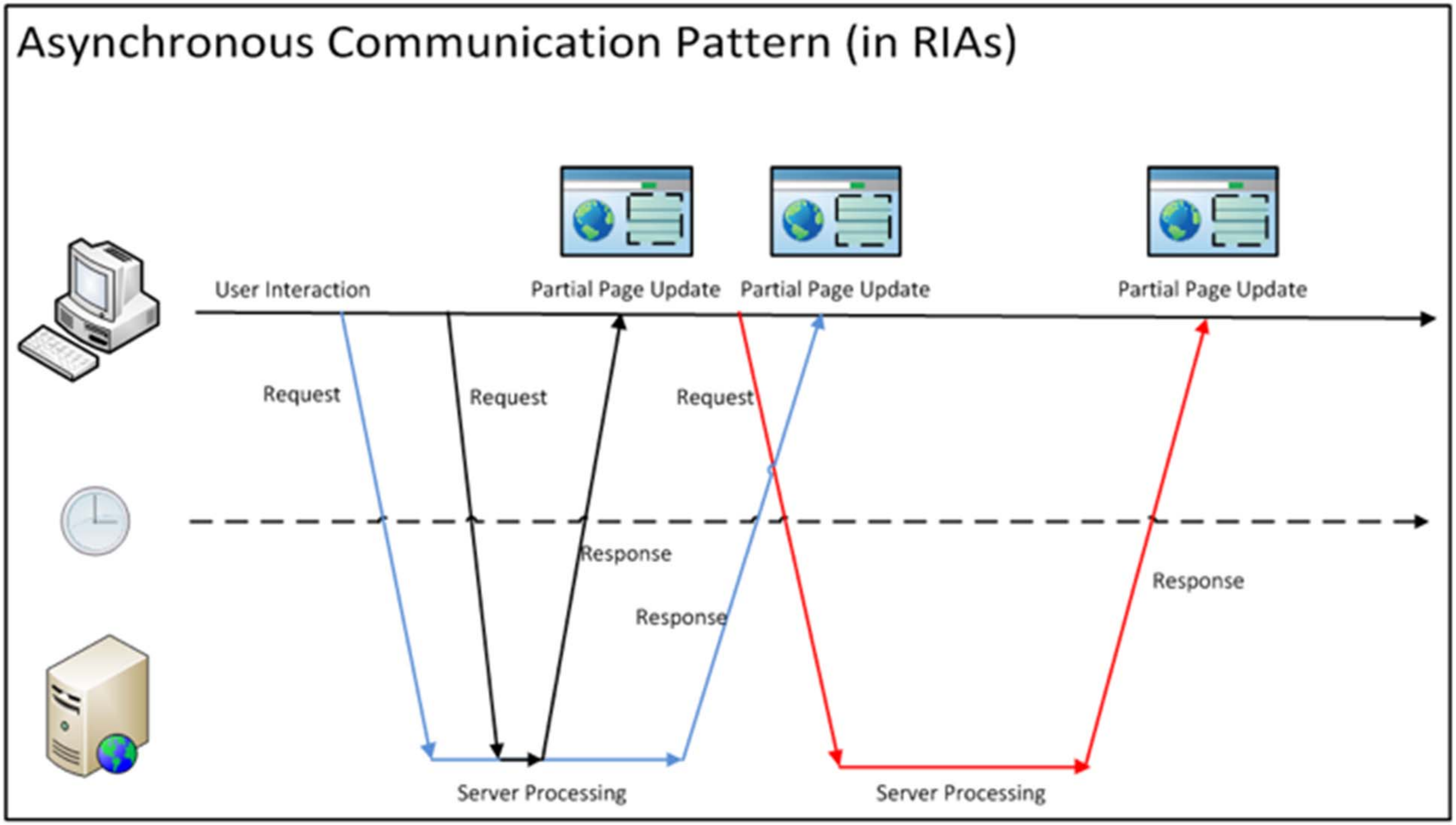
# The evolving Web

- **Traditional Web**
  - **Static web** :  HTML pages identified by an URL
  - **“deep web”** : HTML pages dynamically created by server, identified by URL with parameters

# The evolving Web (ii)

- **Web 2.0 : Rich Internet Applications (RIA)**
  - pages contain executable code (e.g. JavaScript, Silverlight, Adobe Flex...); executed in response to user interactions or time-outs (so-called events); script may change displayed page (the “state” of the application changes) – with the same URL.
  - AJAX: script interacts asynchronously with the server to update the page.

# Example of interactions

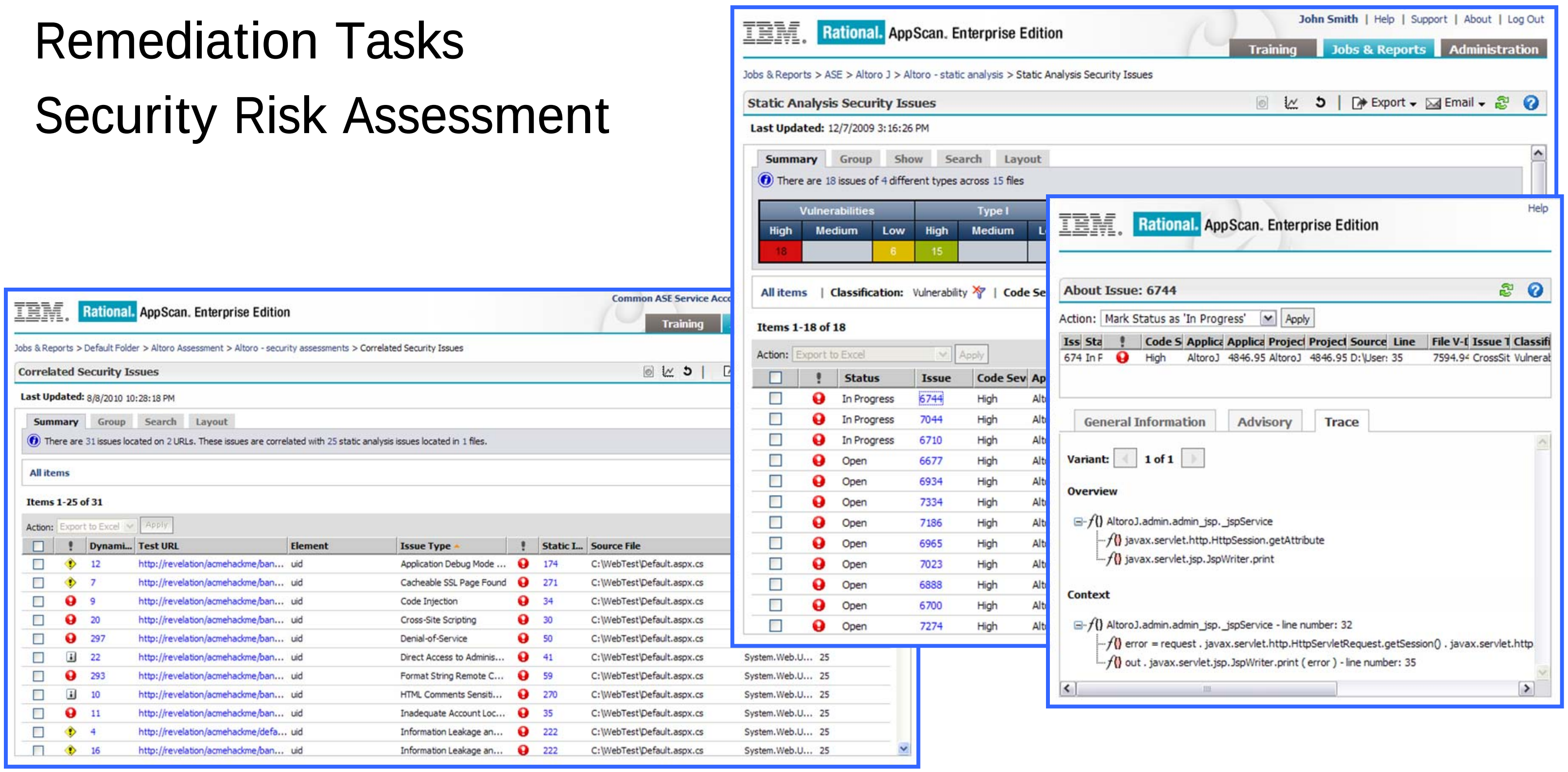


# Why crawling

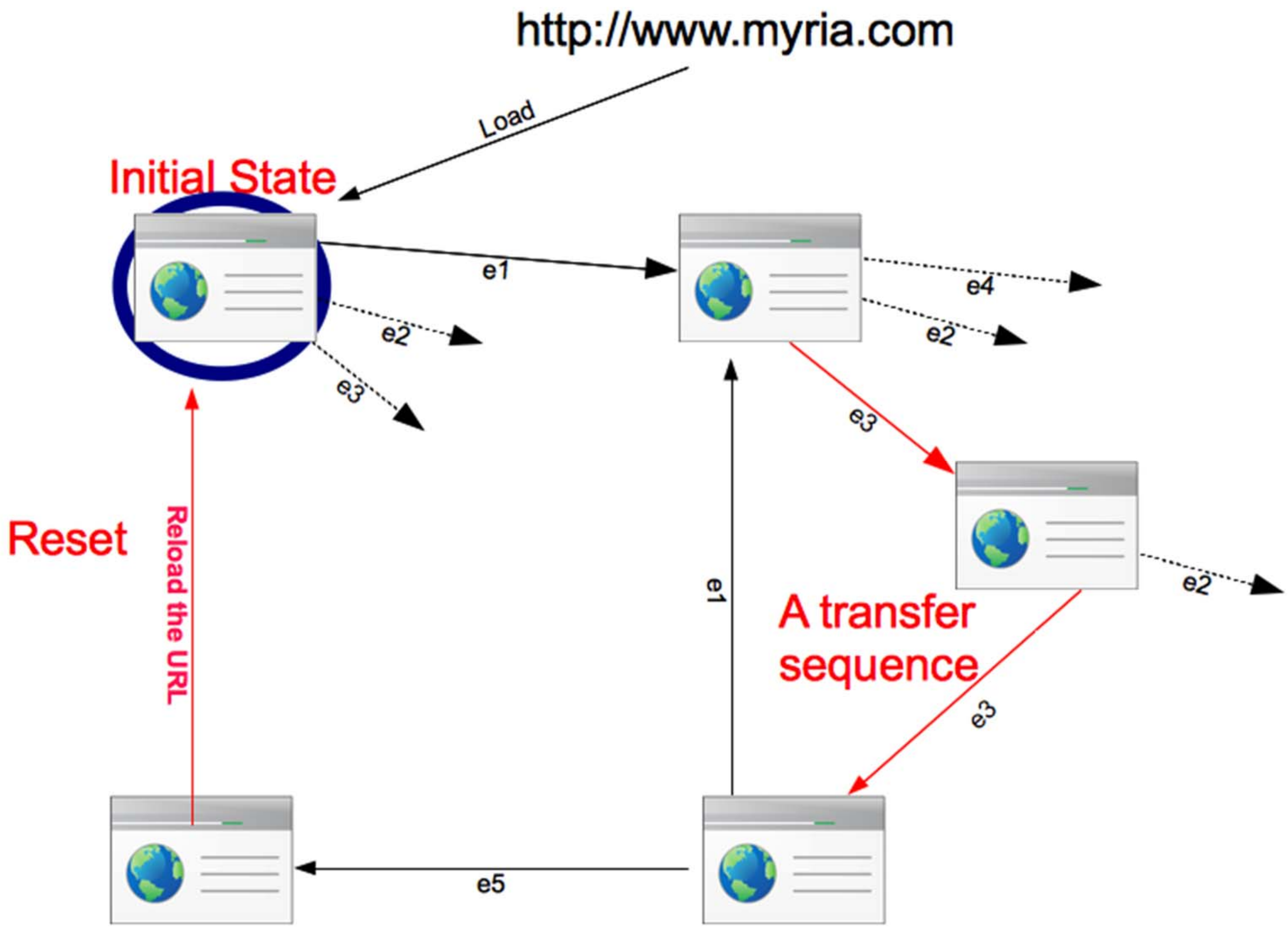
- Objective A: find all (or all “important”) **pages**
  - for content indexing
  - for security testing *(this is of interest to IBM)*
  - for accessibility testing *(this is of interest to IBM)*
- Objective B: find all **links** between pages
    - thus building a **graph model** of the application
      - pages (or application states) are nodes
      - links (or events) are edges between nodes
  - for ranking pages, e.g. Google ranking in search queries
  - for automated testing and model checking of the web application
  - for assuring that all pages have been found

# IBM security testing tools

- Security Issues Identified with **Static Analysis** (white-box view)
- Security Issues Identified with **Dynamic Analysis** (black-box view)
- Aggregated and correlated results
- Remediation Tasks
- Security Risk Assessment



# Crawling example



# Difficulties with crawling RIAs

- State identification
  - A state can not be identified by a URL.
  - Instead, we consider that the state is identified by the **current DOM** in the browser.
- Most links (events) do not contain a URL
  - An event included in the DOM , normally, does not identify the next state reached when this event is executed.
  - To determine the next state, we have to execute that event.
    - In traditional crawling, the event (link) contains the URL which identifies the next state reached (without executing the link)
- Accessibility of states
  - Most states are not directly accessible (no URL) – only through “seed” URL and a sequence of events (and intermediate states)

# Important consequence

- For a complete crawl (a crawl that ensures that all states of the application are found), the crawler has to execute all events in all states of the application
  - since for any of these events, we do not know, a priory, whether its execution in the current state will lead to a new state or not.
    - Note: In the case of traditional web crawling, it is not necessary to execute all events on all pages; it is sufficient to extract the URLs from these events, and get the page for each URL only once.

# **A theoretical problem:**
# Discover the behavior of a state machine by testing

- Possible approach: Explore all transitions reachable from the initial state.
  - Assumption: Each state provides the list of valid inputs for the transitions from this state.
  - For testing each transition, start with a **reset**.
  - After the execution of a tested transition, execute one sequence of the W-set (and possibly repeat for other W sequences)
- **Problem (in general):** We do not know the W-set.
- **Solution for RIA crawling:** the state is identified by its DOM (actually, we use the hash) *– like using a readState interaction*

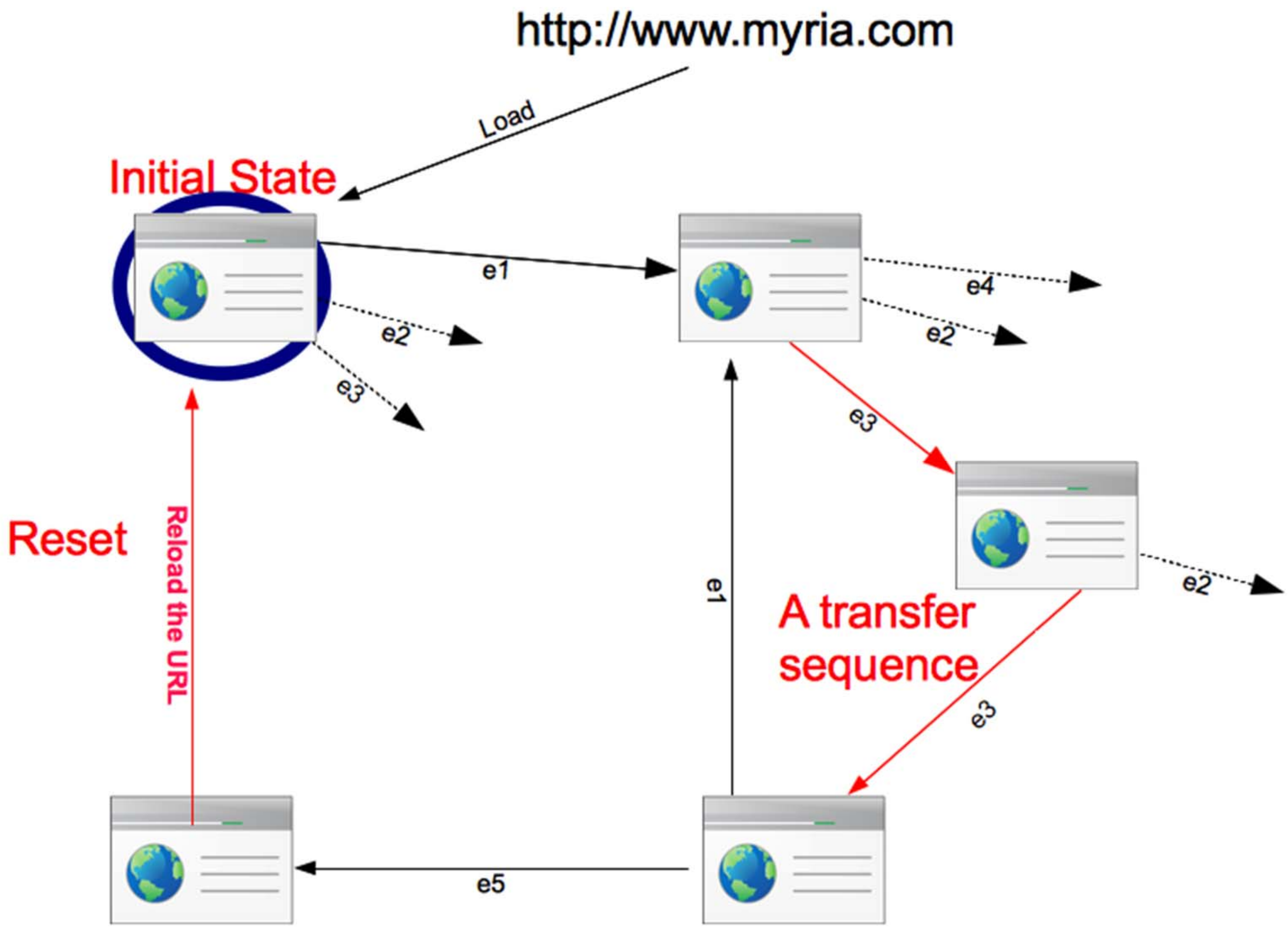
# Crawling Strategies

- The strategy decides what URL/event to be explored next.
- An “efficient” strategy discovers the states as soon as possible *(our definition).*
  - Note: Event executions through intermediate states and resets normally dominate the crawl time. – We want to reduce this as much as possible

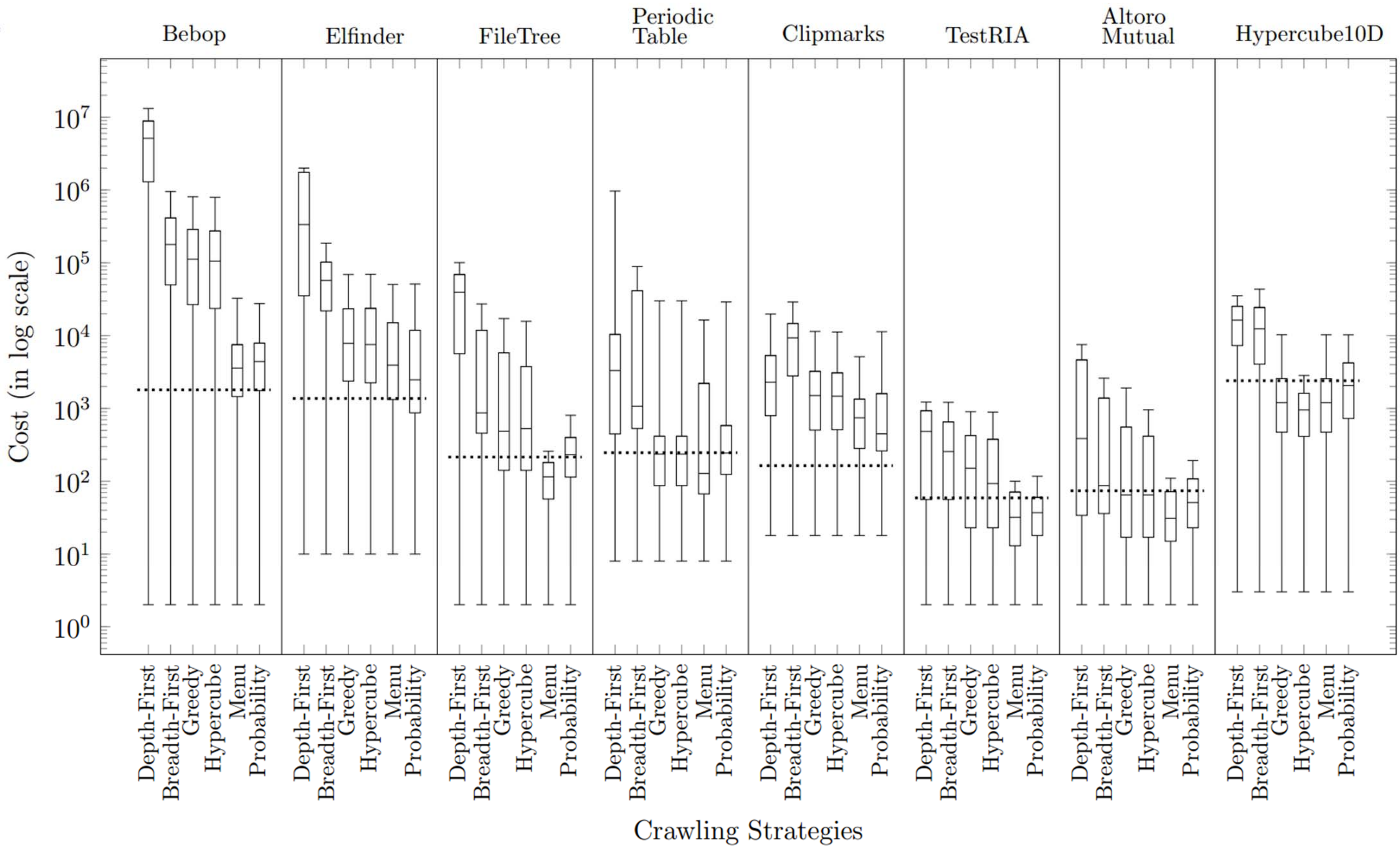
# Examples of Crawling Strategies

- Breadth
- Depth first
- Greedy: finds shortest path through the explored application graph to a node with a non-executed transition
- Model-Based Crawling *(has been proposed by our group)*
  - Hypercube
  - Menu Model
  - Probability

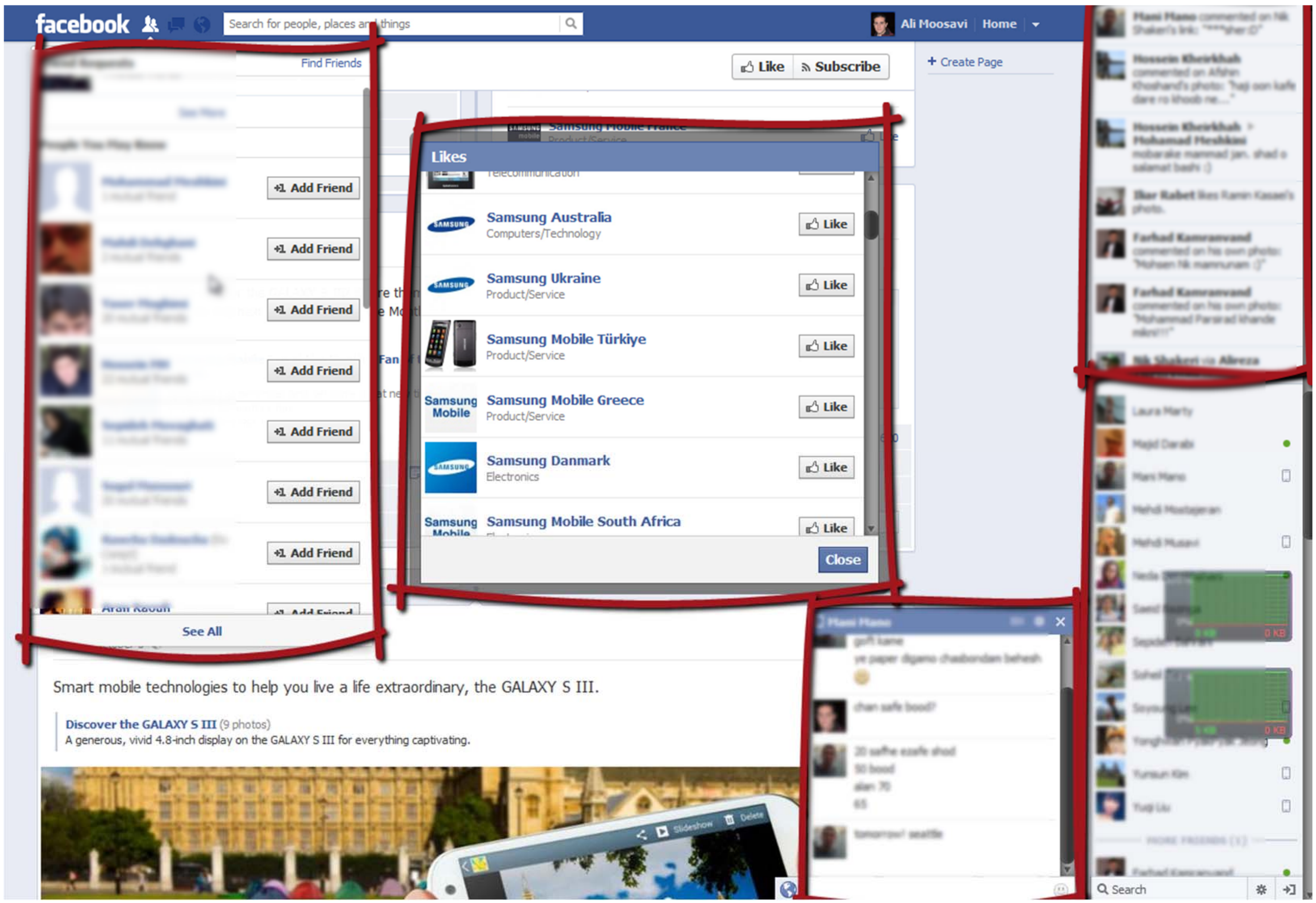
# Crawling example



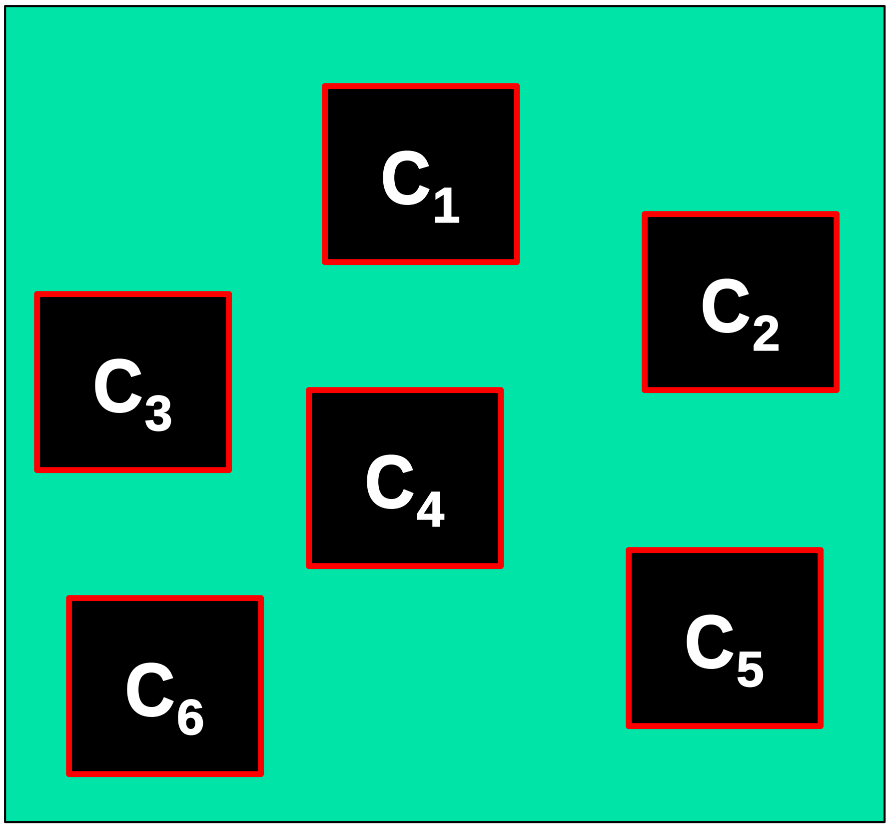
# Performance of crawling strategies



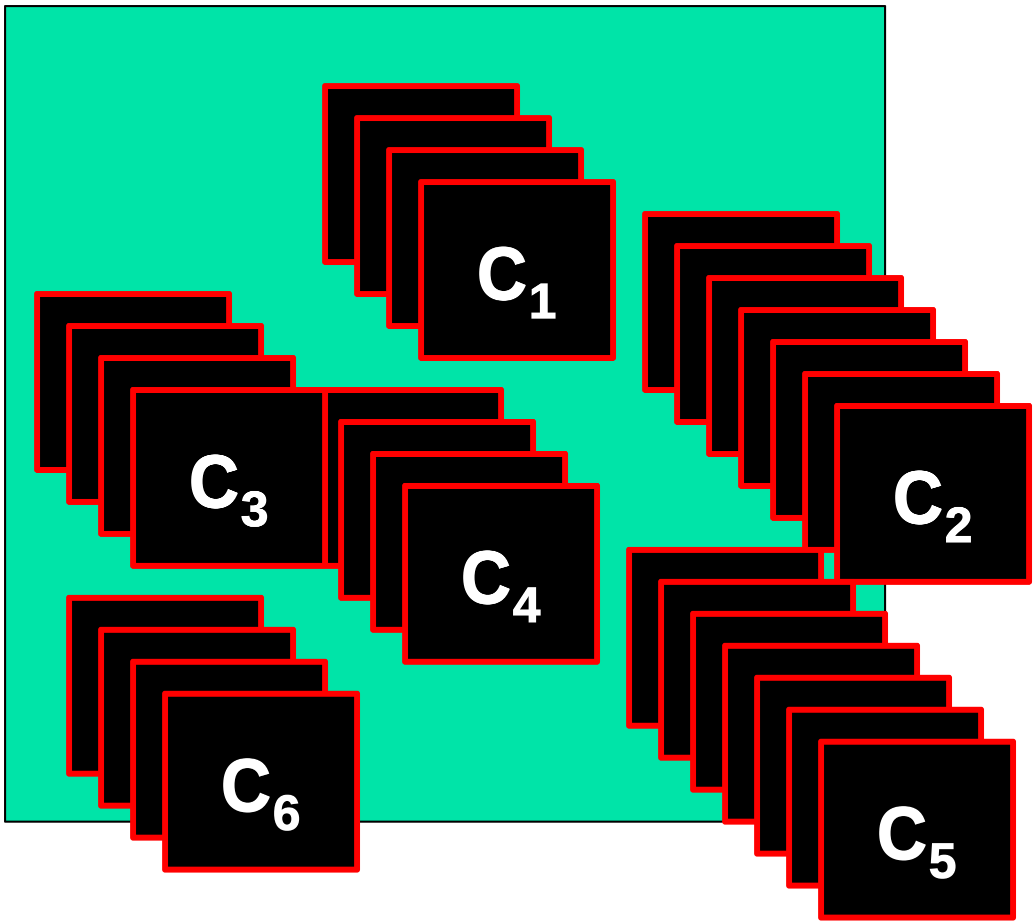
# Component-based crawling



# Abstract view



# Intrinsic Limitations




*k* components

$$C_1, C_2, \ldots, C_k$$

$$C_i$$

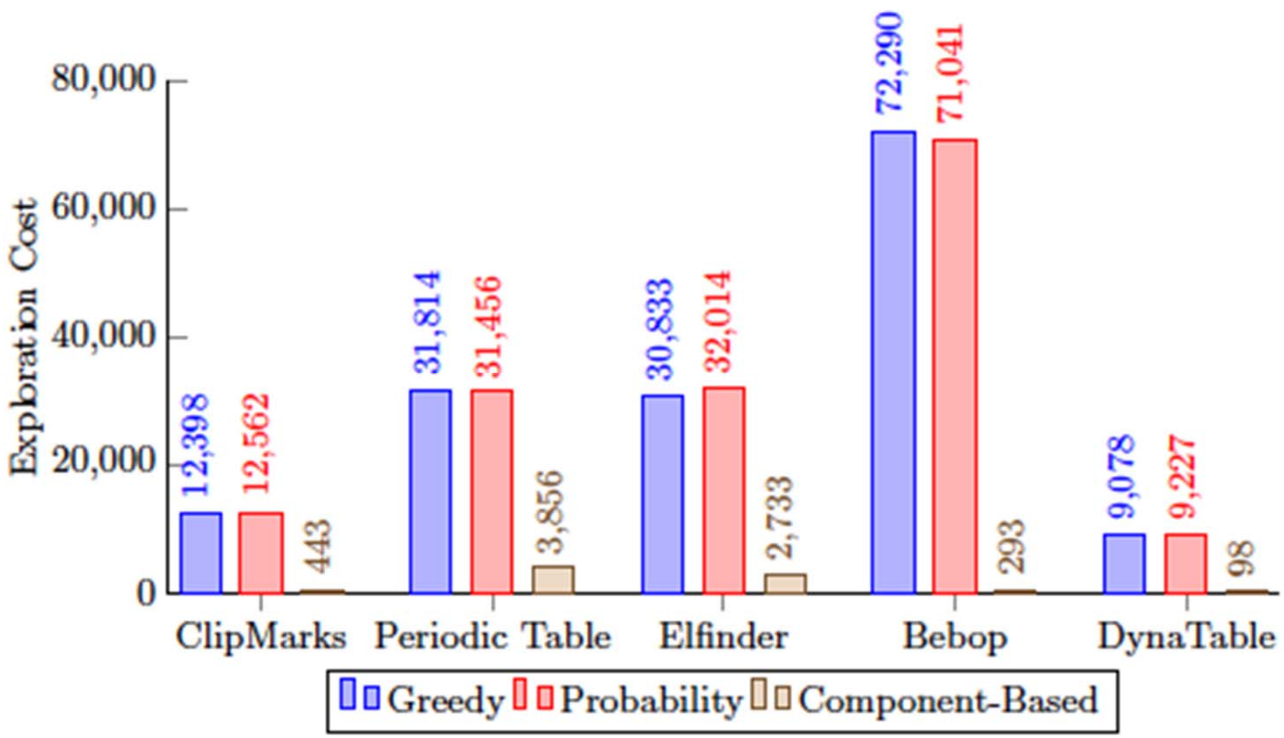
Eacl $\bar{C}_i$ ıponent

has component-
states

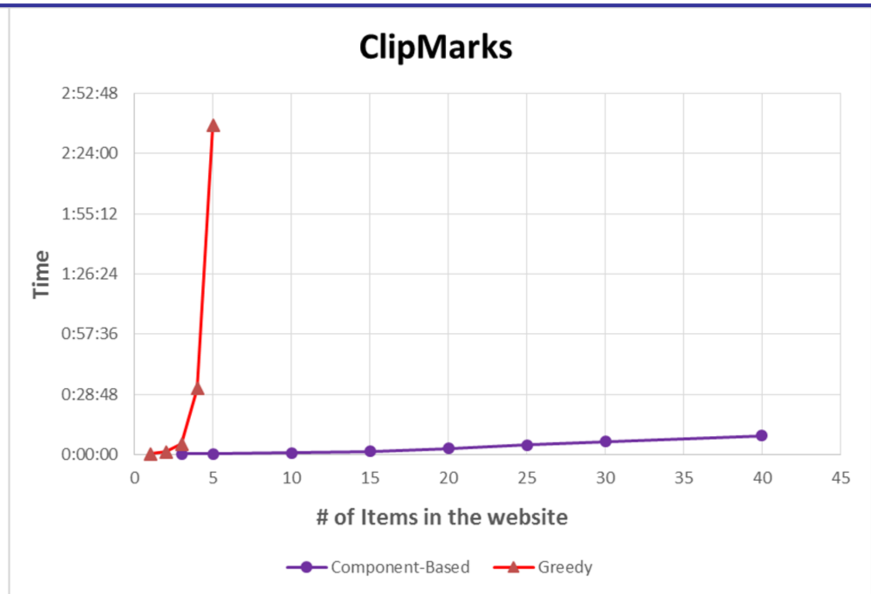
# Idea of component-based crawling

- Partition the DOM into independent components (types)
- Each component has a set of component states (instances)
- Crawl all component instances of a given component independently of other components

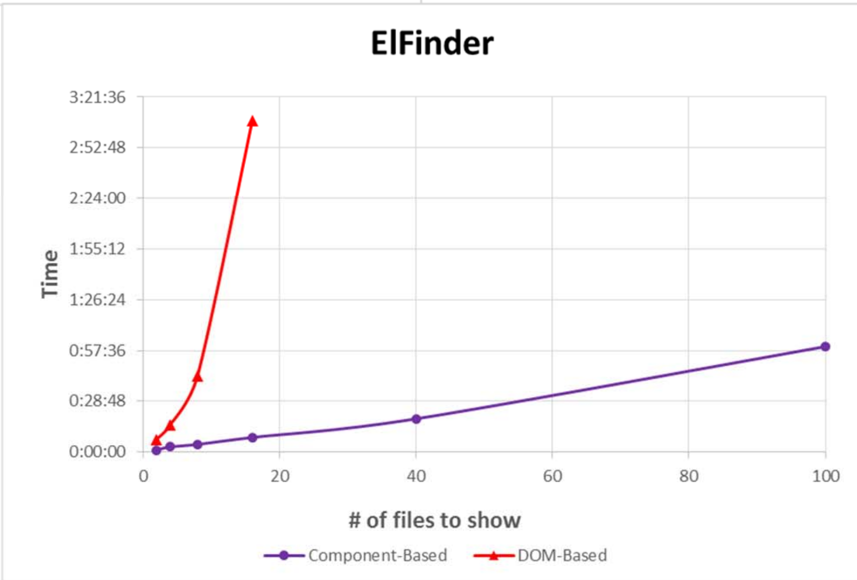
# Results – for small RIAs



# Component-based crawling has good scalability


**But no coverage guarantee**



# Conclusions

- For reactive systems, state machine models can often be used to represent important aspects of the behavior.
- There is a long history of model-based testing, especially for state machine models.
- Test coverage considerations can be based on the IUT (white-box testing) or on the specification (black-bock testing). How to evaluate test coverage does not depend on this question, but on the language used to define the behavior which is being tested :
  - (a) state machine testing methods (e.g. W-method), or
  - (b) coverage criteria for program behavior.
  - Both approaches should be combined for testing Extended State Machine models.
- It is not clear whether the test coverage guarantees provided by state machine testing methods are important in practice.
- Discovering the behavior of a black-box state machine by testing – is this a new problem waiting for a solution ? – I doubt that it is practically relevant, though.
  - If the machine supports a readState input, the well-known Greedy algorithm can be used for this purpose, as we do for RIA crawling.

# Further readings


1. **Some notes on the history of protocol engineering** ( G. v. Bochmann, D. Rayner and C. H. West ), Computer Networks journal, 54 (2010), pp 3197–3209.
2. **Formal methods in communication protocol design** ( G. v. Bochmann and C. A. Sunshine ), (invited paper) IEEE Tr. COM-28, No. 4 (April 1980), pp. 624-631, reprinted in "Communication Protocol Modeling", edited by C. Sunshine, Artech House Publ., 1981
3. **Protocol specification for OSI** ( G. v. Bochmann ), Computer Networks and ISDN Systems 18 (April 1990), pp.167-184
4. **Synchronization and specification issues in protocol testing** ( B. Sarikaya and G. v. Bochmann ), IEEE Trans. on Comm., COM-32, No.4 (April 1984), pp. 389-395.
5. **Generating synchronizable test sequences based on finite state machines with distributed ports** (Luo, G., Dssouli, R., Bochmann, G.v., Ventakaram, P., Ghedamsi, A.: ), in Proceedings of the IFIP Sixth International Workshop on Protocol Test Systems, Pau, France, September 1993, pp. 53–68 (1993)
6. **Test selection based on finite state models** ( S. Fujiwara, G. v. Bochmann, F. Khendek, M. Amalou and A. Ghedamsi ), IEEE Transactions on Software Engineering, Vol.17, no.6, June 1991, pp. 591-603
7. **Principles and methods of testing finite state machines – a survey** ( Lee, D., Yannakakis, M. ), Proceedings of the IEEE 84(8), 1089–1123 (1996)
8. **FSM-based incremental conformance testing methods** ( K. El-Fakih, N. Yevtushenko and G. v. Bochmann ), IEEE Trans. on SE, Vol. 30, 7 (July 2004), pp. 425-436.
9. **Automatic executable test case generation for extended finite state machine protocols** ( C. Bourhfir, R. Dssouli, E. Aboulhamid, N. Rico ), Proc. 10th International Workshop on Testing of Communicating Systems, 1997, Cheju Island, Korea, pp. 75-90.
10. **Test generation with inputs, outputs, and quiescence** (G. Tretmans), in Tools and Algorithms for Construction and Analysis of Systems, Second International Workshop, TACAS '96. Springer-Verlag, 1996, pp. 127–146.
11. **Testing systems specified as partial-order input/output automata** ( G. v. Bochmann, S. Haar, C. Jard and G. V. Jourdan ), Proc. IFIP Testcom/FATES Workshop, Tokyo, June 2008, LNCS.
12. **Fault models for testing in context** ( A. Petrenko, N. Yevtushenko and G. v. Bochmann ), in Proc. IFIP symposium FORTE-PSTV'96, Formal Description Techniques IX, R. Gotzhein and J. Bredereke, Chapman and Hall, 1996, pp. 163-178.
13. **Fault diagnosis in extended finite state machines** ( K. El-Fakih, S. Prokopenko, N. Yevtushenko and G. v. Bochmann ), Proc. TestCom 2003 - the IFIP 15th International Conference on Testing of Communicating Systems, May 2003 in Sophia Antipolis, France, LNCS 2644, Springer Verlag, pp. 197-210.
14. **Multiple fault diagnostics for finite state machines** ( A. Ghedamsi, G. v. Bochmann and R. Dssouli ), Proc. IEEE INFOCOM'93, San Francisco, USA, March 93
15. **Improved Usage Model for Web Applications Reliability Testing** ( B. Wan, G. v. Bochmann and G. V. Jourdan ), Proc. 23th IFIP Int. Conf. on Testing Software and Systems (ICTSS'11), Paris, Nov . 2011
16. **Using logic to solve the submodule construction problem** ( G. v. Bochmann ), Journal on Discrete Event Dynamic Systems, Springer, January 2012, pp. 1 - 13.
17. **On the realizability of collaborative services** ( H. N. Castejòn, G. v. Bochmann and R. Braek ), Journal of Software and Systems Modeling, Vol. 10 (12 October 2011), pp. 1-21.
18. **Model-Based Rich Internet Applications Crawling: "Menu" and "Probability" Models** (Choudhary, S., Dincturk, E., Mirtaheri, S., Bochmann, G. v., Jourdan, G.-V., and Onut, V. ), Journal of Web Engineering, 13(3&4), pp. 243 – 262, 2014

# Thanks !

Any questions

or comments ??

For copy of slides, see

**http://www.site.uottawa.ca/~bochmann/talks/testing.ppt**