
MDV, April 2010

Some Modeling Challenges when Testing Rich Internet Applications for Security

Kamara Benjamin, Gregor v. Bochmann
Guy-Vincent Jourdan, Vio Onut

Introduction

- Web-based applications are very common, more are designed and deployed every day
- Their architecture, their exposure, the components involved, the ease of development make web-application frequent targets for attacks
- One tool in the security toolbox is the web app security scanner

Introduction

- Rich Internet Applications:
 - Client code modifies the DOM
 - Asynchronous communication between client and server
 - Variety of client side component (JavaScript, Silverlight, Adobe Flex...) running concurrently
- ⇒ We have lost the ability to “crawl” these applications.

Introduction

- Without crawl, we cannot
 - Index a site
 - Perform some automatic tests
 - Scan for security, accessibility, usability....
- ⇒ We need to create a new model suitable for RIAs, that can restore our ability to crawl Web Applications

Background and Assumptions

- The RIAs that we are analysing:
 - Actions are repeatable: we can “reset” the application so that sending the same input in the same order and at the same time will produce the same output.
 - The only source of non determinism is concurrency.
 - Application are “user-input” free: every interaction between the application and the user can be modeled as a choice between a known finite set of possibilities.

Background and Assumptions

- The model that we are building:
 - Eventually uncovers all the client states
 - Contains the events that lead from one state to another
 - Is built deterministically (repeatably)
 - Is built “efficiently”: interesting information is found early

Background and Assumptions

- Comparing states:
 - During a crawl, we will repeatedly uncover the “same” state
 - Often, “same state” does not mean equality
 - Depending on the goal of the crawl, the notion of state equality may be different
- ⇒ We need a flexible state equivalence function on which our crawling algorithm relies.

Existing Ajax Crawling Solutions

There are some existing RIA crawlers (most if not all of them being Ajax crawlers), but they have shortcomings for our goals:

- Incomplete models:
 - Typically just attempt one ordering of concurrent events
 - Fail to take into account the intermediate states reached by the client after sending an asynchronous message and before receiving the corresponding response.

Existing Ajax Crawling Solutions

There are some existing RIA crawlers (most of not all of them being Ajax crawlers), but they have shortcomings for our goals:

- Efficiency strategy:
 - Crawls can be very long/infinite
 - We must define what is “interesting” in the model and attempt to uncover it first

Existing Ajax Crawling Solutions

There are some existing RIA crawlers (most of not all of them being Ajax crawlers), but they have shortcomings for our goals:

- State equivalence:
 - A realistic crawler cannot rely on equality for deciding if a state has been visited before
 - Tools that integrate this concept seem to use distances instead of equivalence functions
 - Tools do not seem to separate clearly the “state equivalence” decision from the crawling algorithm (perhaps Crawljax does it, ICST2010)

A Strategy (or a direction for one)

- One key goal of the crawling algorithm is to converge towards a complete model, while finding the “good parts” as early as possible in the process.
- To do so, it is necessary to state what reasonable assumptions can be made regarding what information is more interesting in the model.
- Obviously, these are just reasonable assumptions, and counterexamples will be easy to find.
- We have identified two assumptions that seem reasonable and that we can deal with

A Strategy (or a direction for one)

1. Assumption one: if we have already explored a client state by executing all of its concurrent events at least once, then it is more interesting to explore a state that we haven't explored at all yet rather than exploring the first state further
 - A given state with n concurrent events could lead to $n!$ ways of executing the events
 - We will need to explore the state completely eventually, but if we know of another state which is less explored, then the latter gets priority

A Strategy (or a direction for one)

2. Assumption two: if we have a state $s1$ in which we have already executed all the events in a few different orders, executing a subset of the events that has not been executed yet is more interesting than executing a subset of the events that was already executed but in a different order.
- The emphasis is on discovering new states, not confirming new transitions between known states

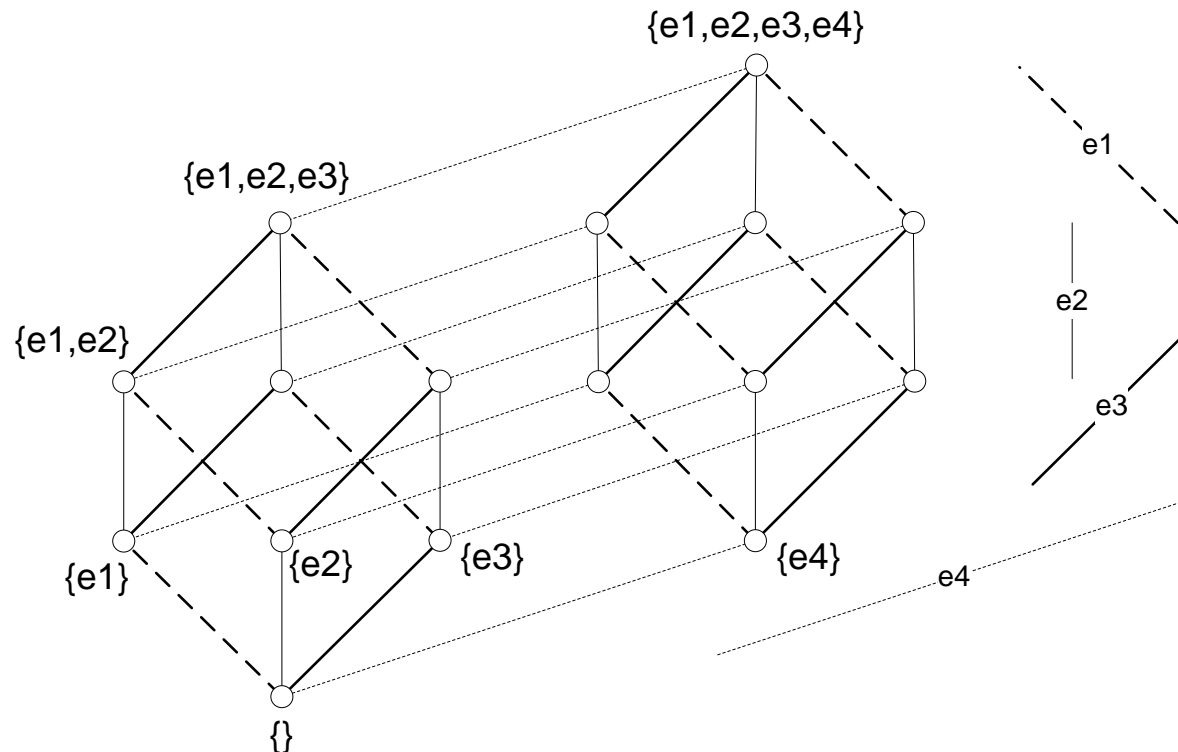
A Strategy (or a direction for one)

A crawling strategy compatible with assumption one is easy to define: always give priority to states than have less known information.

- ❑ An adapted in-depth first strategy will succeed
- ❑ Set a limit to the depth in order to avoid getting stuck in infinite looping paths

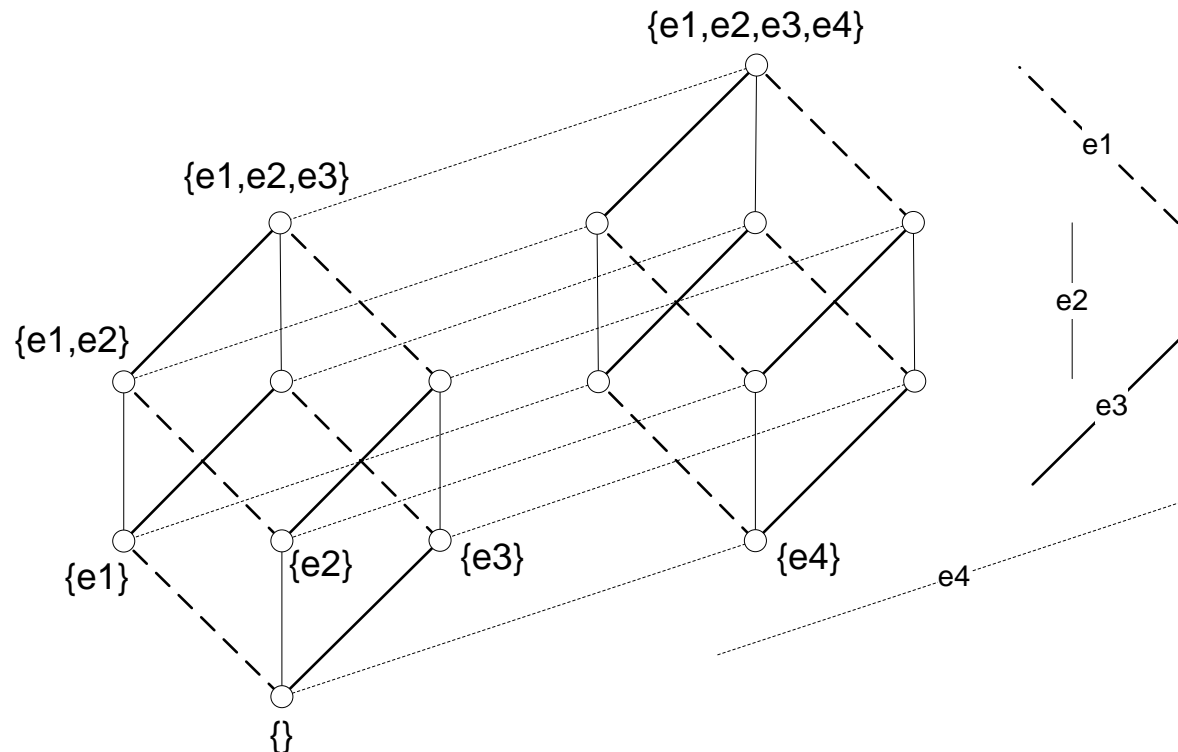
A Strategy (or a direction for one)

A crawling strategy compatible with assumption two is trickier. Assume that a state has n concurrent events. It can be seen as a hypercube:

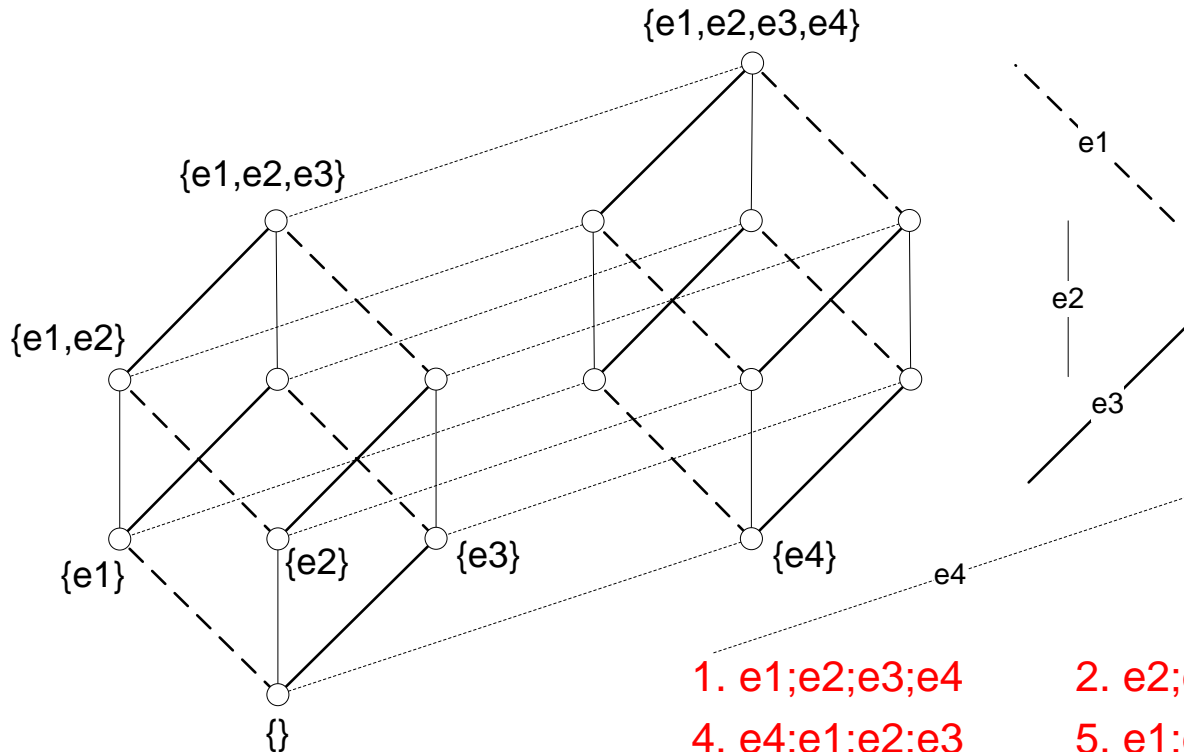


A Strategy (or a direction for one)

In order to crawl w.r.t. assumption two, we need to find the 2^n states as early as possible in the $n!$ paths



A Strategy (or a direction for one)



1. e1;e2;e3;e4

2. e2;e3;e4;e1

3. e3;e4;e1;e2

4. e4;e1;e2;e3

5. e1;e3;e4;e2

6. e2;e4;e1;e3

7. e3;e1;e2;e4

8. e4;e2;e3;e1

9. e1;e4;e2;e3

10. e2;e1;e3;e4

11. e3;e2;e4;e1

12. e4;e3;e1;e2

13. e1;e2;e4;e3

14. e2;e3;e1;e4

15. e3;e4;e2;e1

16. e4;e1;e3;e2

17. e1;e3;e2;e4

18. e2;e4;e3;e1

19. e3;e1;e4;e2

20. e4;e2;e1;e3

21. e1;e4;e3;e2

22. e2;e1;e4;e3

23. e3;e2;e1;e4

24. e4;e3;e2;e1

A Strategy (or a direction for one)

This is a minimal chain decomposition of the hypercube. It can be done in w chains (and this is the best), where w is the width of the order (Dilworth, 1950)

In the case of an hypercube of dimension n , the width is $\text{Choose}(\text{Ceil}(n/2), n)$.

Thus we must enumerate $n!$ chains, the first $\text{Choose}(\text{Ceil}(n/2), n)$ of which go through the 2^n states.

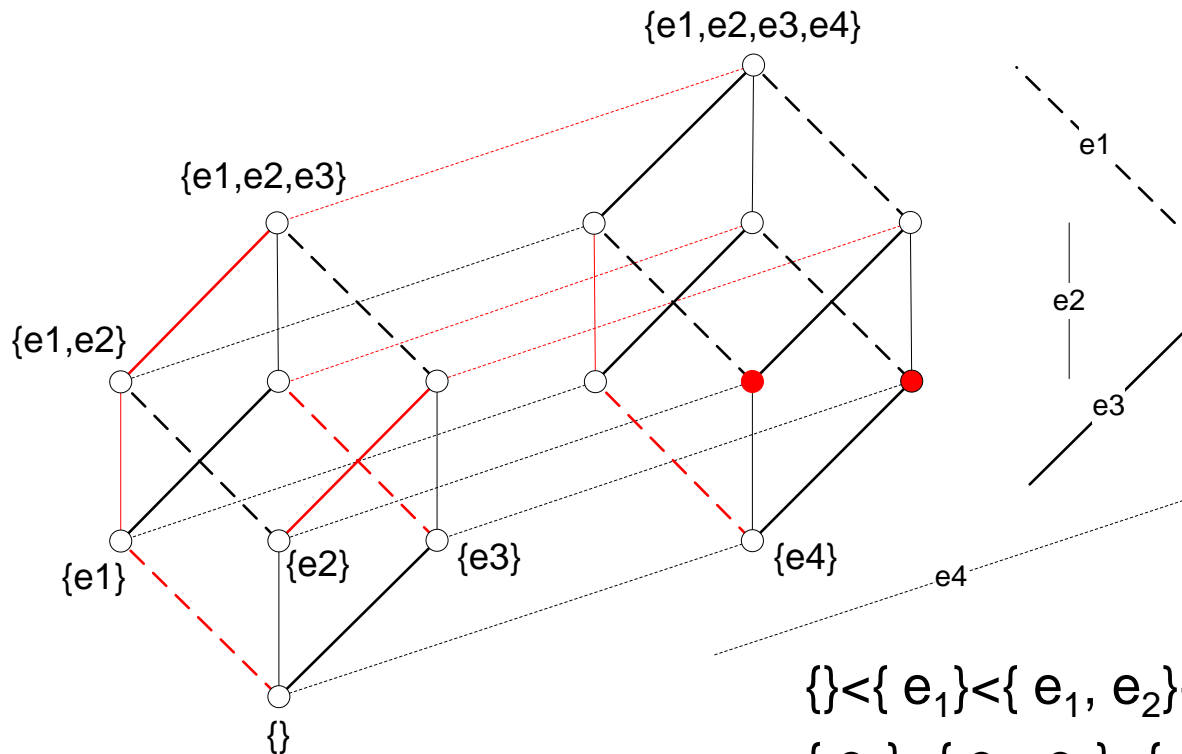
A Strategy (or a direction for one)

We can use a *canonical symmetric chain decomposition* (CSCD) of a hypercube for this. Based on de Bruijn, Tengbergen, and Kruyswijk:

- The CSCD of a hypercube of size 0 contains the single chain (\emptyset) .
- For $n \geq 1$, the CSCD of a hypercube of dimension n contains precisely the following chains:
 - For every chain $A_0 < \dots < A_k$ in the CSCD of a hypercube of dimension $n - 1$ with $k > 0$, the CSCD of a hypercube of dimension n contains the chains:
$$A_0 < \dots < A_k < A_k \cup \{n\} \text{ and } A_0 \cup \{n\} < \dots < A_{k-1} \cup \{n\}.$$
 - For every chain A_0 of size 1 in the CSCD of a hypercube of dimension $n - 1$, the CSCD of a hypercube of dimension n contains the chain:

$$A_0 < A_0 \cup \{n\}$$

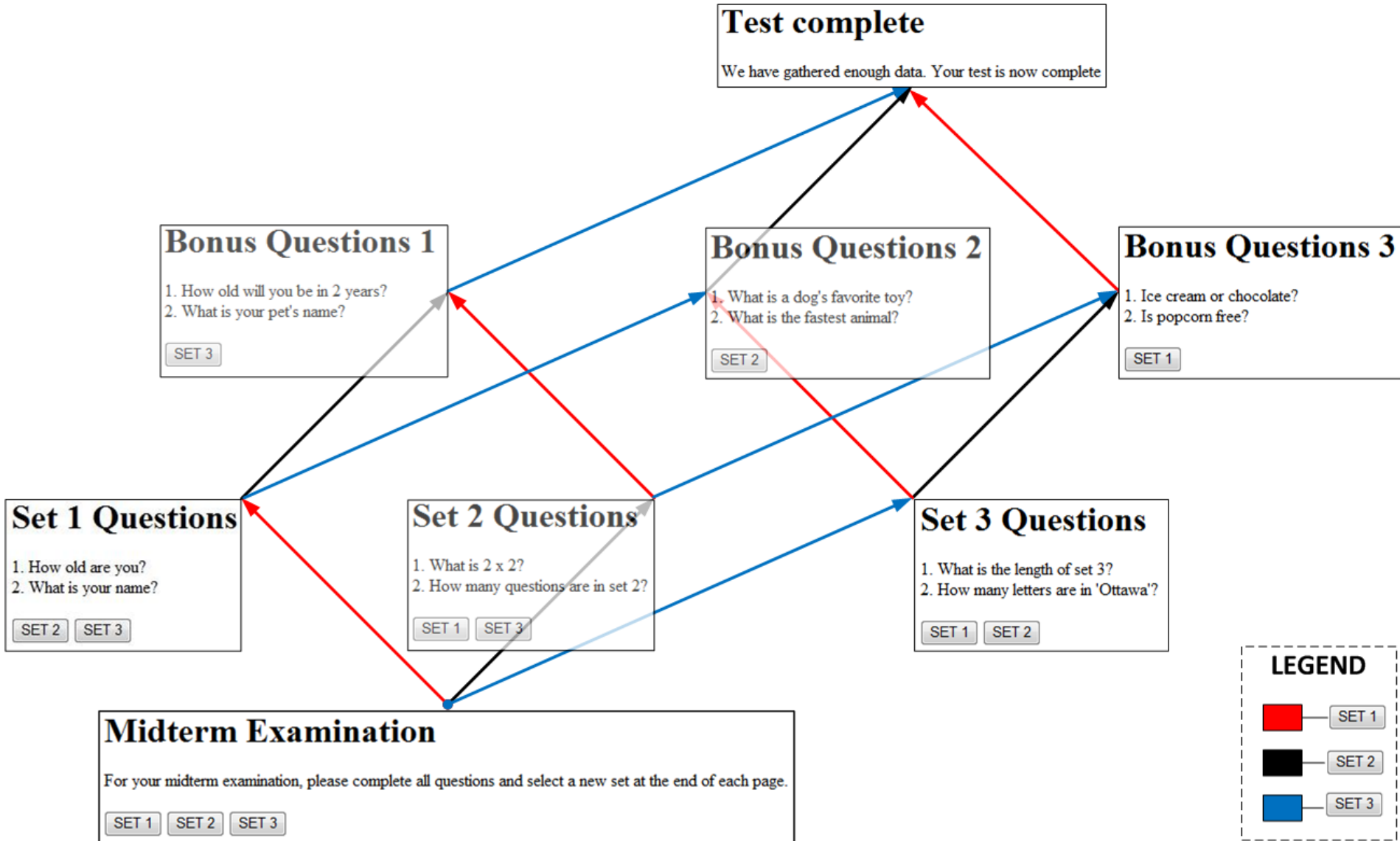
A Strategy (or a direction for one)



$\{\} < \{e_1\} < \{e_1, e_2\} < \{e_1, e_2, e_3\} < \{e_1, e_2, e_3, e_4\}$
 $\{e_4\} < \{e_1, e_4\} < \{e_1, e_2, e_4\}$
 $\{e_3\} < \{e_1, e_3\} < \{e_1, e_3, e_4\}$
 $\{e_3, e_4\}$
 $\{e_2\} < \{e_2, e_3\} < \{e_2, e_3, e_4\}$
 $\{e_2, e_4\}$

Prototype Demo

Prototype Demo



Difficulties Ahead Of Us

Many problems are still ahead, including

- Statelessness of Server
- Data Input Values
- State equivalence definition (fast, accurate, compatible with crawling at least)
- Modelling for security?
- ...

Next Steps

- Actual implementation of a complete crawling strategy that satisfies assumption one
- Extension of our hypercube coverage strategy to finish up the $n!$ paths
- Effective strategy for other behaviour, any combination of
 - Appearing events
 - Disappearing events
 - Unexpected state merge
 - Failed expected state merge