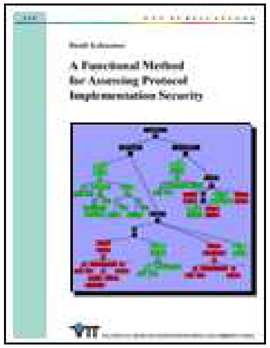
PROTOS - Mini-simulation method for robustness testing

This and more available from:

http://www.ee.oulu.fi/research/ouspg/protos



Described in licentiate thesis of Rauli Kaksonen:

“A Functional Method for Assessing Protocol Implementation Security”. VTT Publication 448

ISBN 951-38-5873-1, (2001)

Canonical URL: http://www.inf.vtt.fi/pdf/

Motivation

- Robustness in face of unexpected (malign) input is a key issue in software security
- Robustness can be tested
  - Traditional (conformance) testing does not probe robustness
  - Complementary approaches are needed
- Our mini-simulation approach was designed for robustness testing
  - Specification-based and flexible (syntax) test design yields effective test material

Every interface has a language

- Explicit (documented)
  - Formally specified -> parser code generated?
- Implicit (not documented)
  - Use the source or analyser, Luke
- Hidden languages foster vulnerabilities
  - E.g. in buffer overflow vulnerabilities data may be interpreted as code
- Syntax testing exercises these languages

Traditional black-box testing vs. robustness testing

1. Does SW deliver required features (conformance)?	1. Does SW fail when exposed to malicious input?
2. Mostly expected input (clean syntax testing)	2. Mostly exceptional and thus unexpected input (dirty syntax testing)
3. Expected outcome for specific input	3. Mostly ignore responses (no oracle) 1 + 1 = 3 is fine
4. “Hundreds of test cases”	4. “Thousands of test cases”

Mini-simulation testing phases

1. Write or acquire a formal interface specification
2. Augment with *rules* and simplify, if needed
3. Design *valid-cases*
4. Define or reuse *anomalies*
5. Insert the anomalies
6. Design test cases
7. Generate test cases
8. Execute tests
9. Analyse the results

1. Specification

- Our tool uses context-free grammar (BNF) with extensions
  - Context-free description of PDU structures may come from BNF or ASN.1
  - After mini-simulation augmentations we have a higher-order attribute grammar
  - From formal specification we can calculate complexity

Example: TFTP (RFC 1350)

```
 2 bytes     string    1 byte     string   1 byte
 ------------------------------------------------
| Opcode |  Filename  |   0  |    Mode    |   0  |
 ------------------------------------------------
         Figure 5-1: RRQ/WRQ packet
```

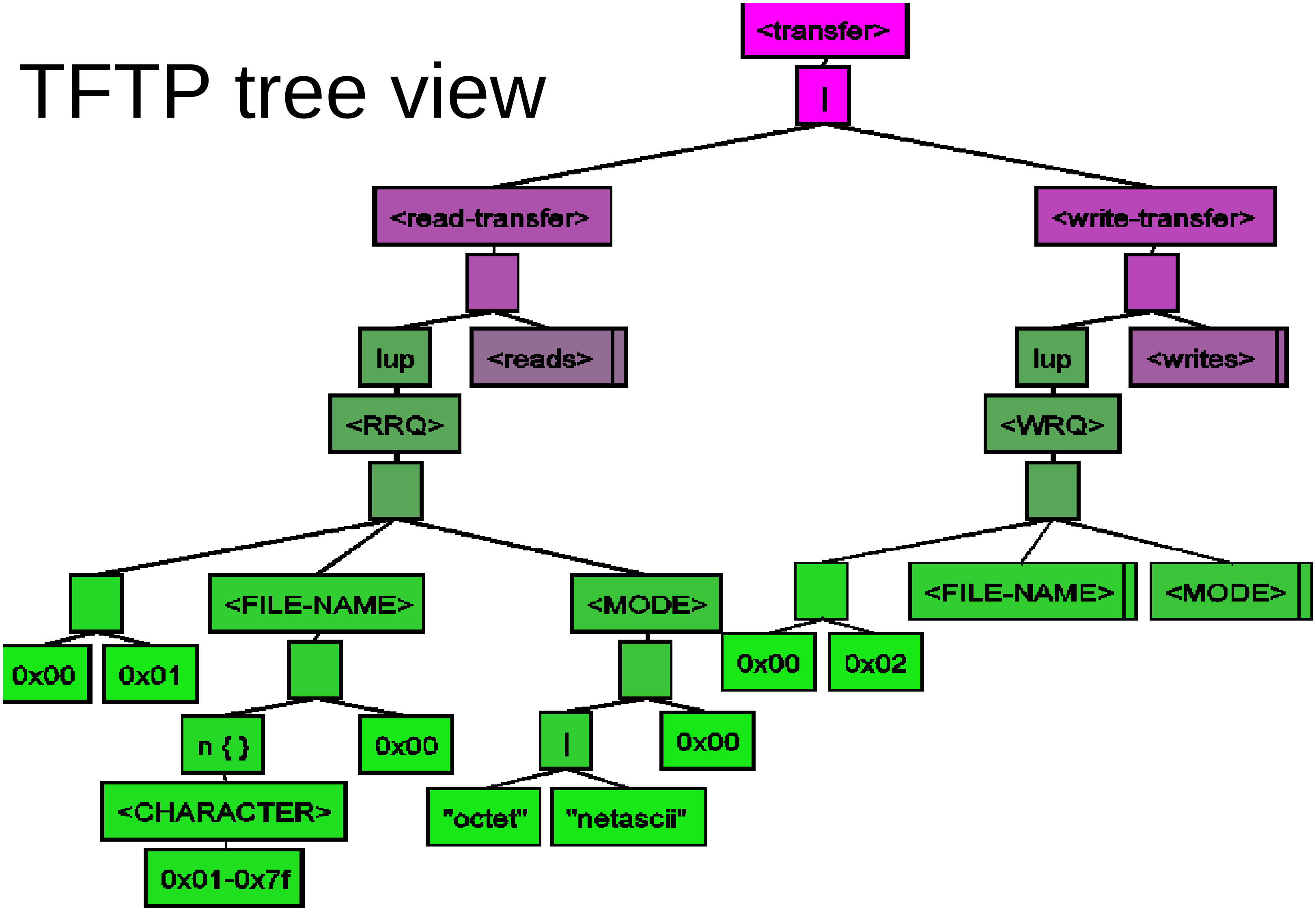
<RRQ> ::= (0x00 0x01) <FILE-NAME> <MODE>
<WRQ> ::= (0x00 0x02) <FILE-NAME> <MODE>

<MODE> ::= ("octet" | "netascii") 0x00
<FILE-NAME> ::= { <CHARACTER> } 0x00
<CHARACTER> ::= 0x01 - 0x7f

2. Rules and simplification

- Our rules are library objects implementing semantics and complex syntax
  - Keeps specification language simple
  - Provides means for communication
  - E.g. checksums, lengths, socket I/O …
- Simplify specification
  - Results a *mini-simulation* with minimal functionality (maximum simplicity) to solve the problem in hand!

TFTP tree view



3. Valid cases

- Create one or more test cases representing valid protocol behaviour
  - Validate understanding of protocol
  - Validate communication with tested software

<RRQ> ::= (0x00 0x01) <FILE-NAME> <MODE>

<MODE> ::= "octet" | "netascii") 0x00
<FILE-NAME> ::= "sample.txt" 0x00

4. Anomalies

- The unexpected elements intended to cause havoc in the tested software
- Designed or reused from anomaly library
- Using the same notation:

<A-string> ::= () | 32x 0x61 0x00 |64x 0x61 0x00 |128x 0x61 0x00 |256x 0x61 0x00 |511x 0x61 0x00 | …

<A-16> ::= 0x00 0x00 |0x00 0x01 |0x00 0x02 |0x00 0x03 |0x00 0x04 |0x00 0x05 |0x00 0x06 |0x00 0xff |0x7f 0xff |0x80 0x00 | …

5. Inserting anomalies

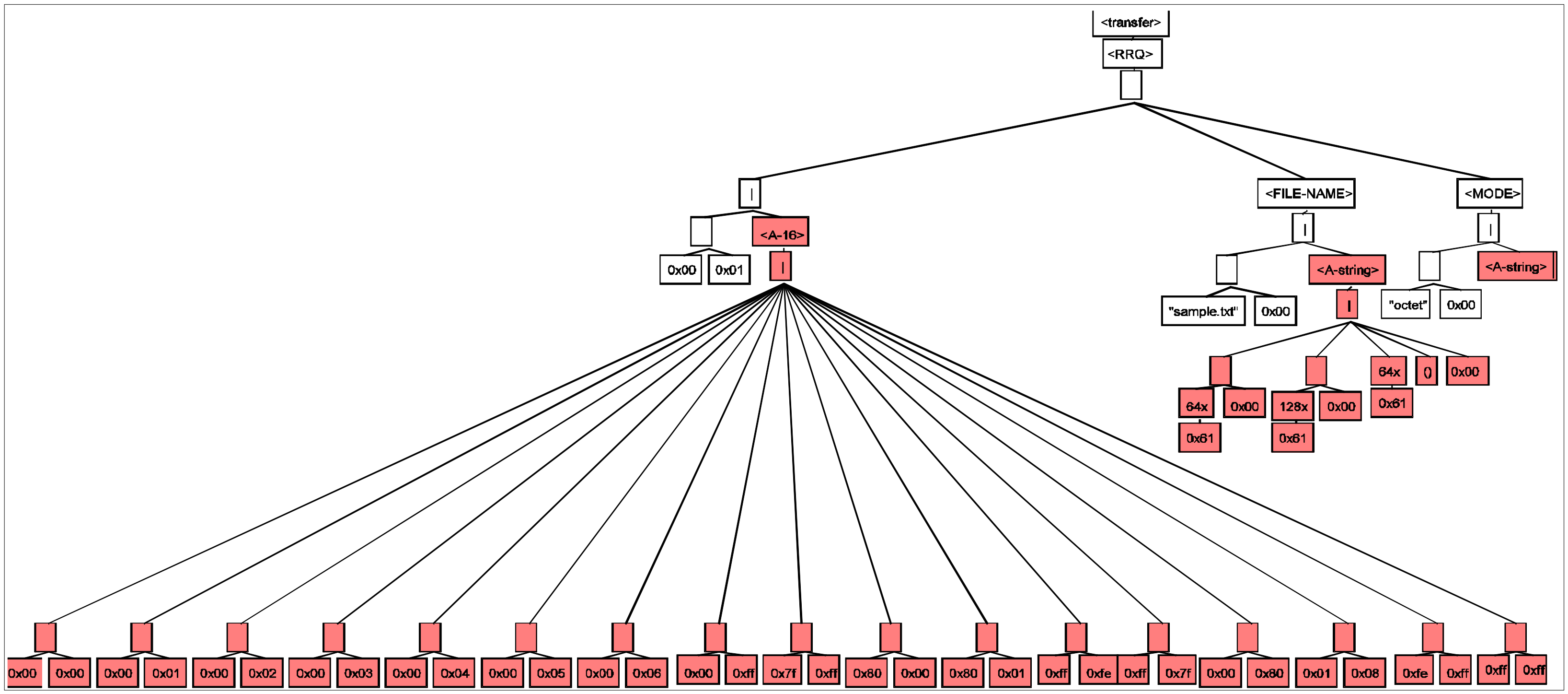
- Anomalies are inserted into specification (with valid cases) as *alternatives*

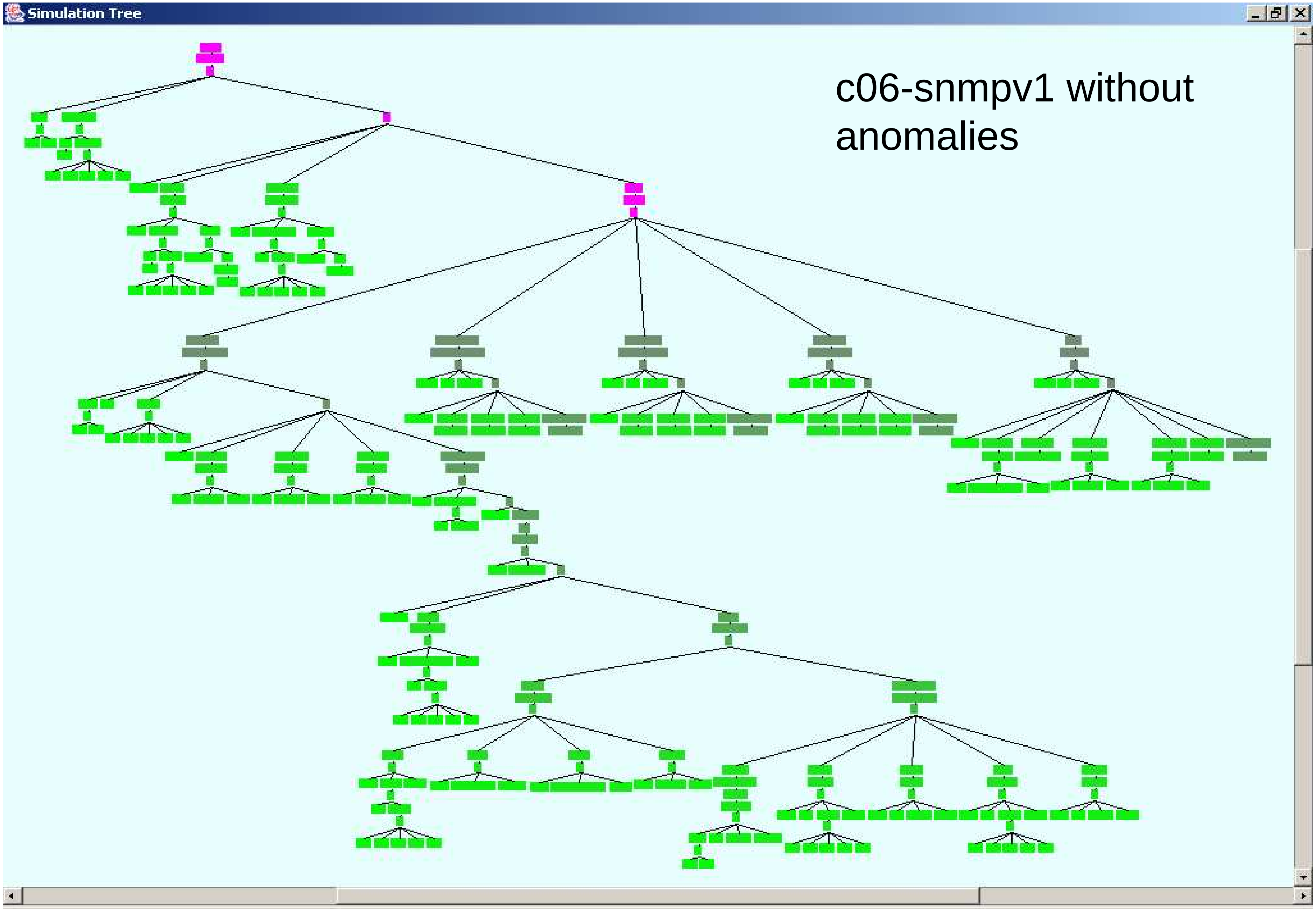
<RRQ> ::= (0x00 0x01 |<A-16>) <FILE-NAME> <MODE>

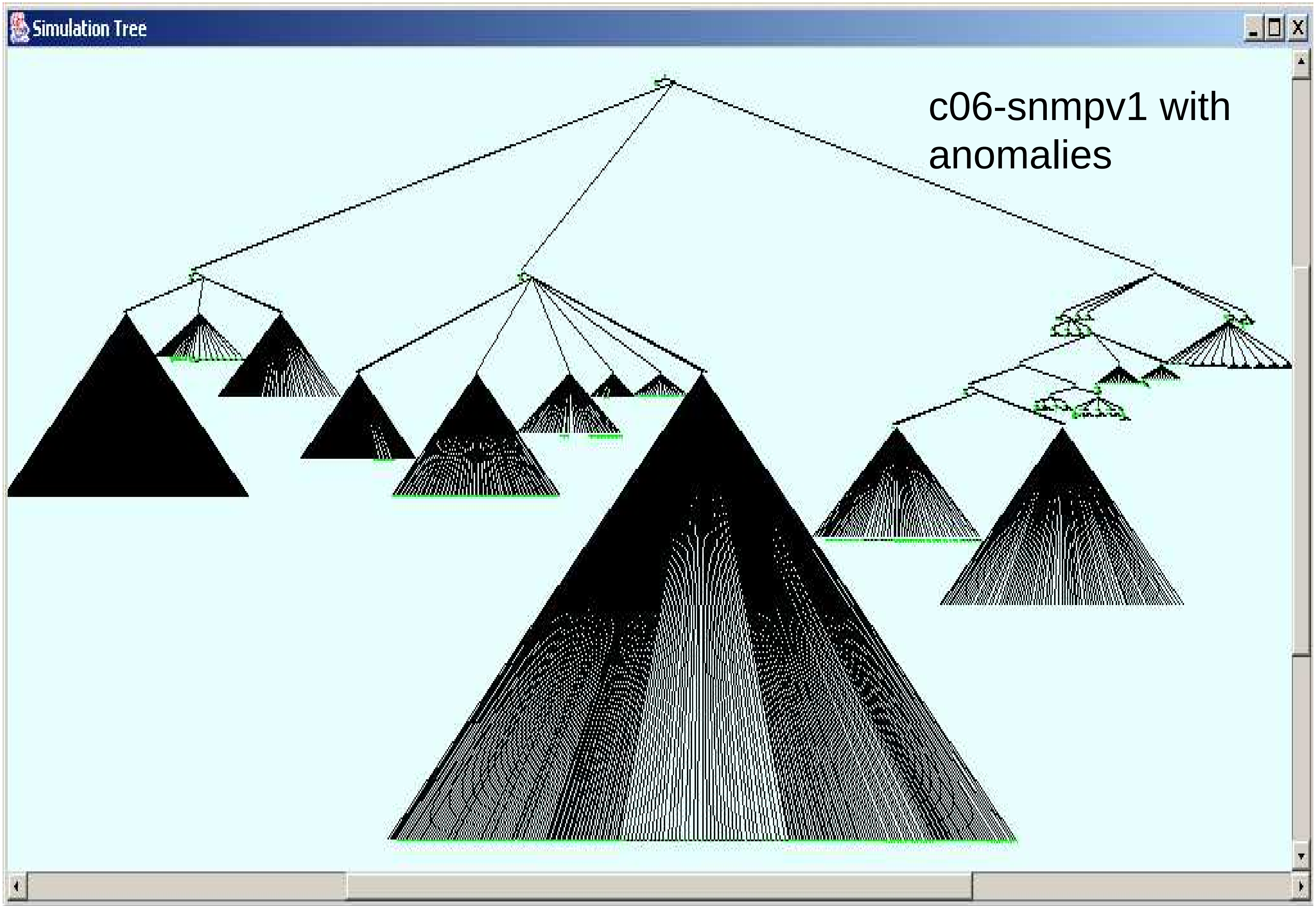
<FILE-NAME> ::= "sample.txt" 0x00 |<A-string>
<MODE> ::= "octet" 0x00 |<A-string>

- In practice the spec is not littered, instead the grammar tree is modified via *replace*

TFTP tree with anomalies




Simulation Tree
c06-snmpv1 without anomalies


Simulation Tree
c06-snmpv1 with
anomalies

6. Designing test cases

- Now we have a mini-simulation for testing
  - With valid cases and anomalies
- Test case design
  - Combine valid elements and anomalies into test cases by instantiating the grammar tree with particular selections
  - E.g. all overflow strings in SNMP get request community name field (a test group)
- Test cases < test groups < test suites

7. Test case generation

- Fully automated
- Results:
  - Binary PDUs (test cases)
  - Test case BNF descriptions
  - Test case documentation
- Since test-cases are relatively cheap to design, generate and execute – you may have plenty of them

8. Execute tests

- For stateless protocols binary-format PDUs (test cases) can be injected into the tested software component directly
- *Preambles* for stateful protocols require more complex handling:
  - BNF-formatted test cases have to be evaluated to mini-simulate the required behaviour
  - Anomalies are injected when in a suitable state

9. Analyse results

- Test case execution log and instrumentation log for the tested software are the basis of the analysis
- Instrumentation sources
  - Debuggers, OS tools, development tools …
- *Valid-case instrumentation* can be used
  - Between test cases a valid case is executed to find out if subject is still alive and kicking

Automation imperative

- Mostly mechanical (man or machine):
  - spec, semantic rules, valid cases, generation, execution and initial analysis
- Heurestic:
  - Anomaly creation and insertion
    - We are working on an anomaly database (e.g. raw.integer.ubit32 replaces unsigned int fields)
  - Test (group) design
    - For now human is needed to arrange infinite input space so that most potent(ial) things come first

Quest for coverage

- Codepath/branch -coverage
  - Could be instrumented via instruction pointer sampling etc.
  - Not in our agenda yet
- Input coverage
  - Comes naturally from the grammar tree approach (e.g. percentage of selections exercised)
  - Work in progress

Ideas that have worked well

1. Modify specification to contain valid cases and anomalies (syntax modelling) + semantic modelling
    - Specification language is all you need
    - Compare to SDL + TTCN + ASN.1
2. Scripting used to modify the specification
    - Original is unmodified and can be maintained separately

Conclusions

- Mini-simulation is alternative for complete system simulation and a viable robustness testing concept
- Simplicity gives us good cost-benefit ratio
  - Automate only mechanical process
- Other potential applications:
  - Stub implementations, debugging aid, analyzers …
- Leave room for human intuition
  - Flexible environment to try new ideas