Performance Analysis with Model Checking*
Extracting performance information from Erlang source code

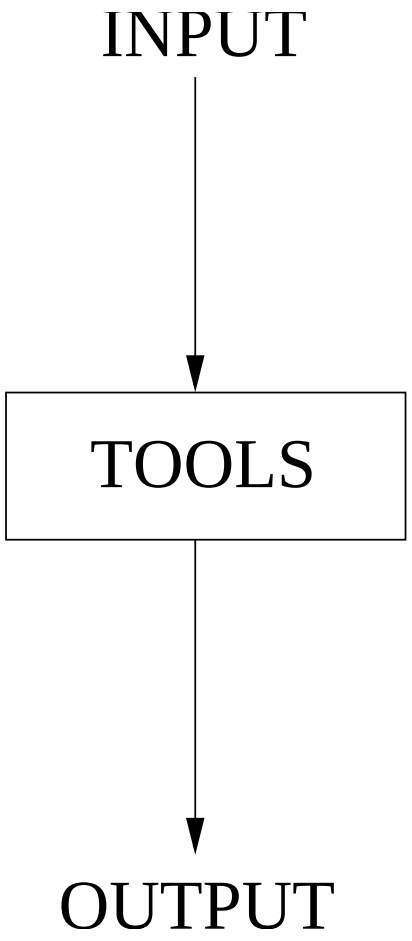
Juan José Sánchez Penas
LFCIA, University of Corunha, Spain
juanjo@lfcia.org

Thomas Arts
IT-university, Göteborg, Sweden
thomas.arts@ituniv.se

EUC 2003. Stockholm, 18th November

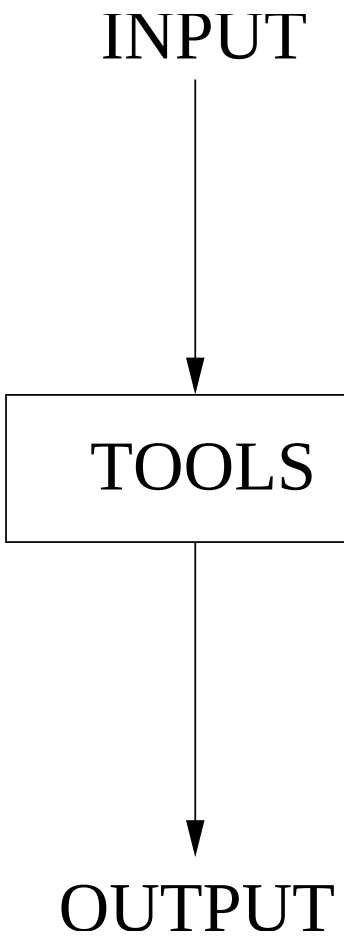
*Work partially supported by MCyT, Spain, Project TIC 2002-02859

General Approach Overview



- Given a real **distributed system**:
  - Functional requirements
  - Design and implementation of the system
  - **Performance requirements**
- Using techniques from **formal methods**:
  - **Model Checking**
  - Theorem Proving
  - **Graph analysis**
- Find and fix:
  - Functional problems
  - Design problems (maintainability, flexibility)
  - **Performance problems**

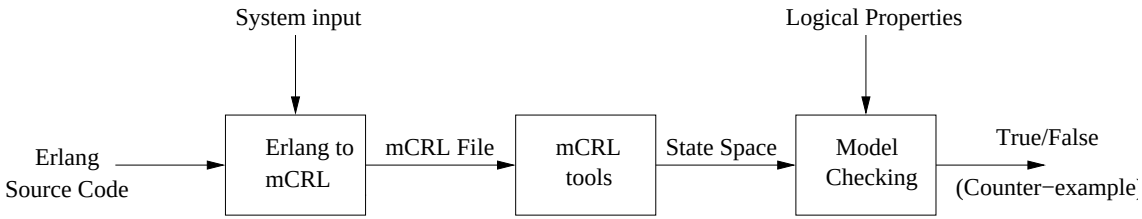
General Approach Overview



- Given:

PERFORMANCE REQUIREMENTS

- Using:



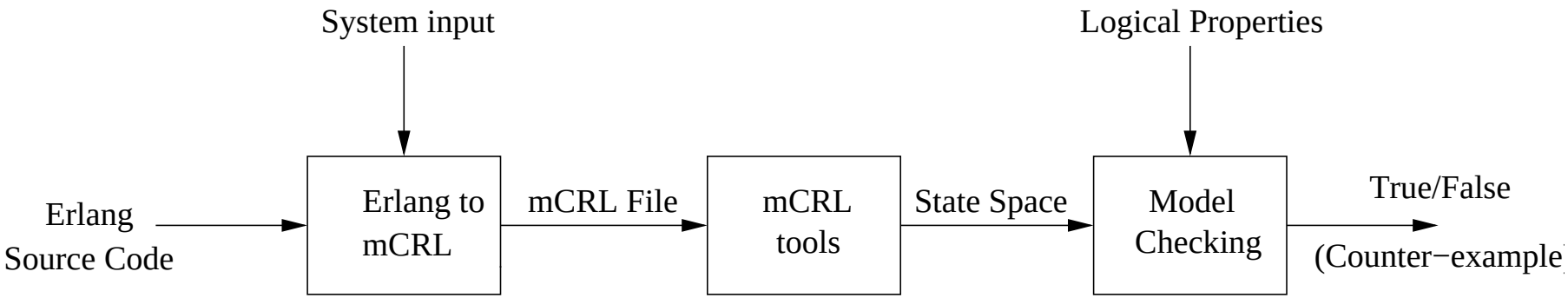
- Find and fix:

CAPACITY INFORMATION

BOTTLENECK INFORMATION

The Proposed Methodology

Existing tools for checking functional properties:



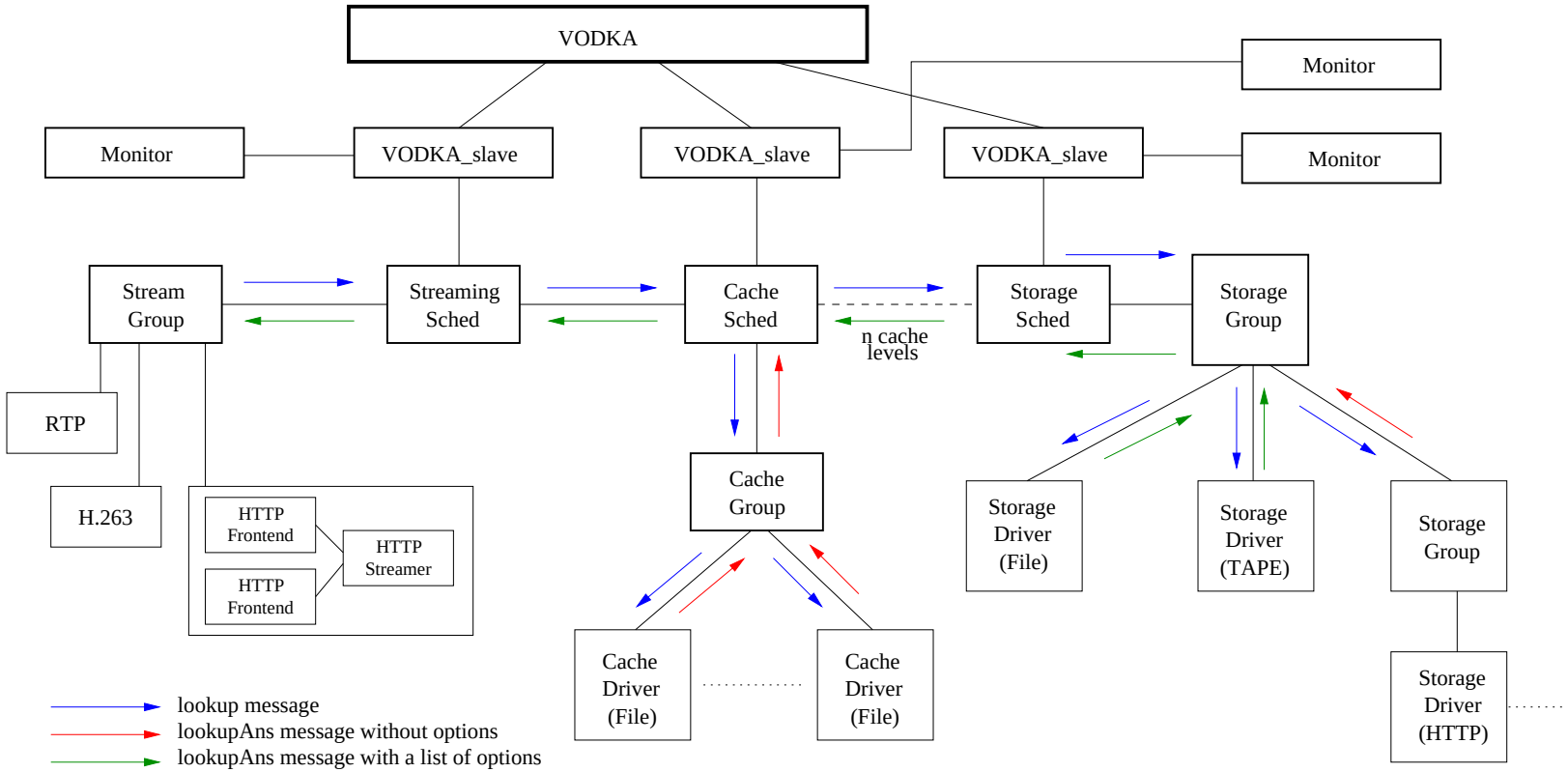
- Generating the full state space of the system **from its configuration**
- Starting directly **from the Erlang source code** of the system (easier with design patterns)
- μCRL as intermediate step (efficient tools for generating state space). Semantics similar to Erlang
- `etomcrl`: Compiler developed before (Thomas Arts and Clara Benac Earle, STTT2003)

Adapted and used for performance properties

Case Study: VoDKA Server - The project

- Hierarchical distributed multimedia server (LFCIA, last 3 years)
- Funded by an European Project and *R*, a Cable Telecommunications Company
- Classical VoD server requirements: Huge storage capacity, high bandwidth, predictable (low) response time, support for a great amount of concurrent users and fault tolerance
- Special requirements for the VoDKA project: Scalability (upwards and downwards), adaptability and low cost
- Hardware: Adaptation of Beowulf Cluster architecture
- Software: Flexible distributed architecture based in Erlang/OTP platform

Case Study: Hierarchical Flexible Architecture



- Flexible architecture based on a hierarchy of specialized **levels**
- Each level is composed by distributed Erlang processes
- Extensive use of *generic server* and *supervision tree* Erlang behaviours

Case Study: Distributed Scheduling

- Completely distributed scheduling subsystem:

no global state, no global decision

- Each process in the scheduling subsystem can implement:
  - Restrictions (number of connections, maximum bandwidth)
  - Scheduling function (filtering, cache algorithms, admission policy)
  - Cost (state of the component and resources still available)

- We want to analyse the system:
  - Information for the 'user' of the system (R) - capacity of the system
  - Information for the designer of the system - how to improve it (bottlenecks)

The Goal: What do we mean by performance analysis?

User point of view

- **Capacity properties** (requirements oriented)
  - System capacity
  - Component capacity
  - Scenario checking

Developer point of view

- **Internal properties** (architecture and protocol analysis)
  - Finding/checking bottlenecks
  - Bottleneck summary from the program graph
  - Extracting/checking message protocol and architecture

The Goal: Problem Explanation by Example

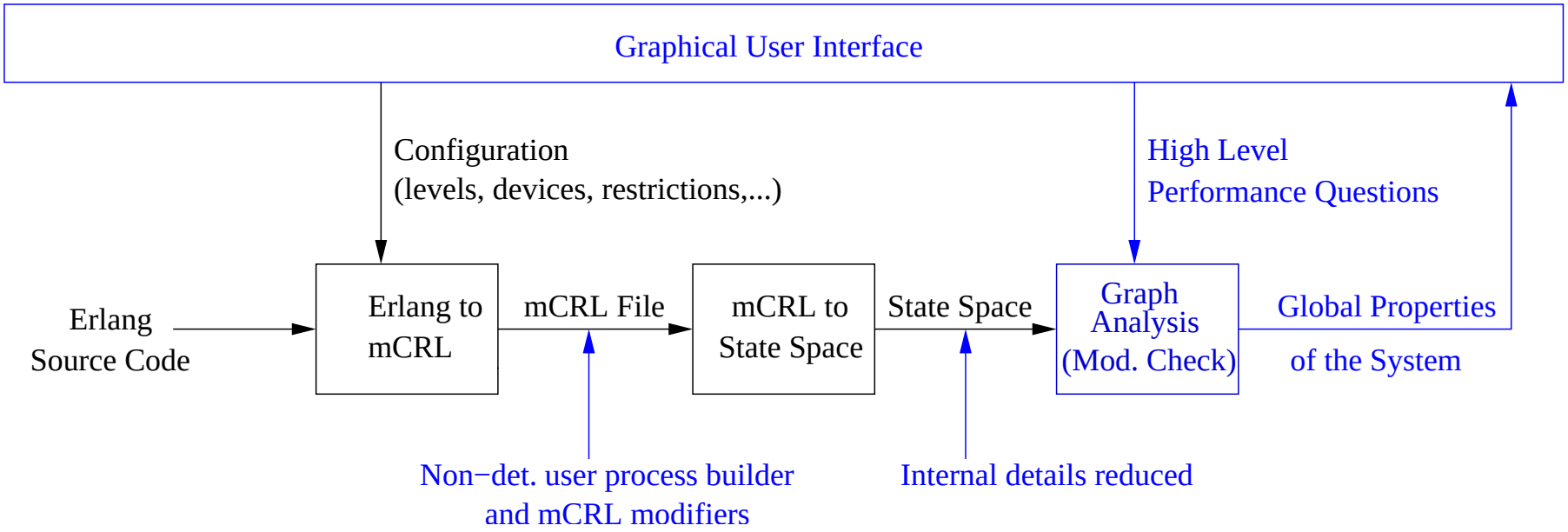
- Goal: Given a *configuration* for the server (the processes, the storage devices, all the restrictions, scheduling functions, and costs):

How can we extract performance information from the source code of the system?

- We want to be able to answer questions like:
  - What is the maximum number of users in the system?
  - What is the minimum number of users such that serving any MO is not possible?

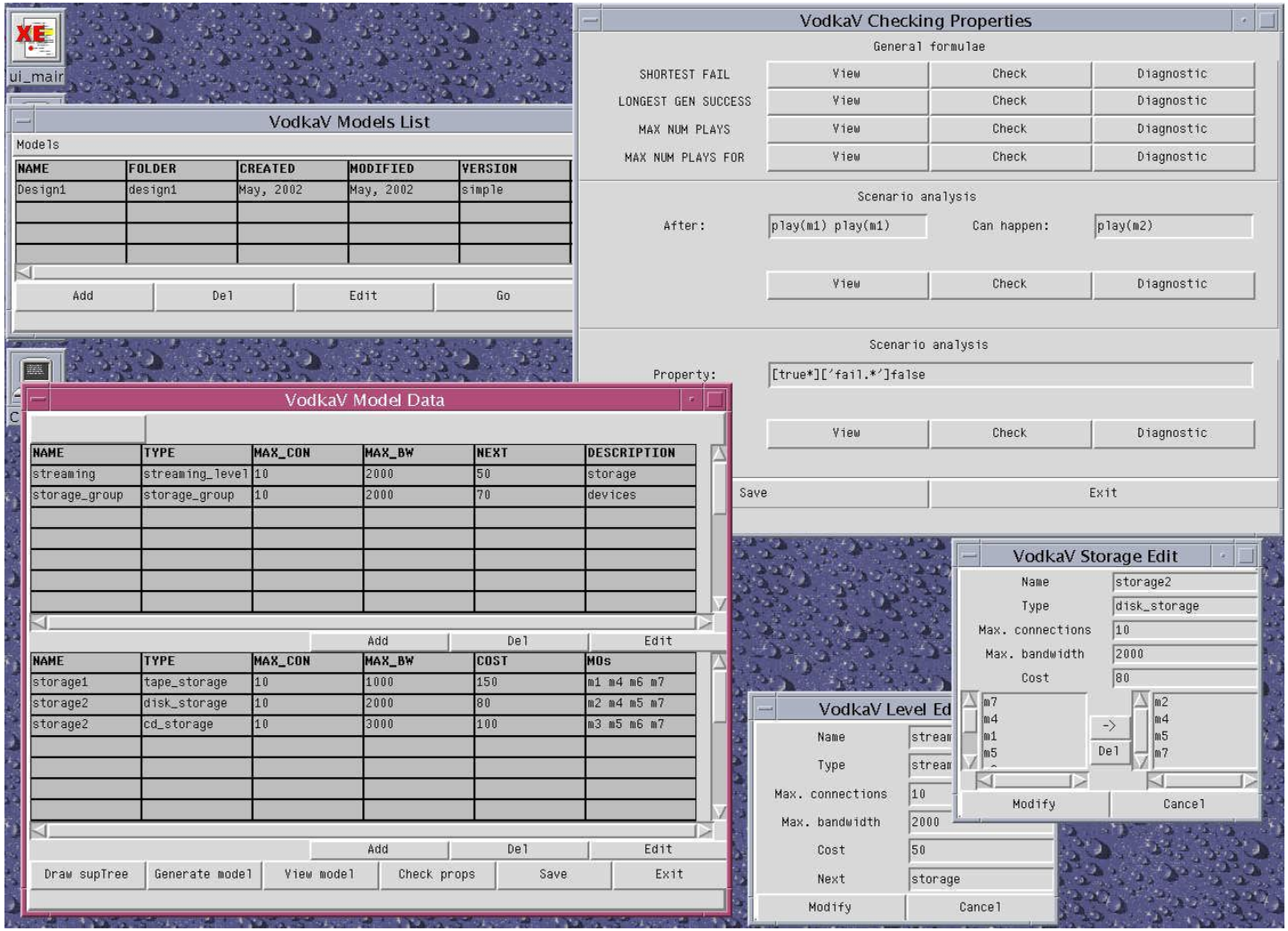
  - Would it be better to move MO from storage1 to storage2?
  - Why (bottleneck) MO cannot be served to N users at the same time

The Proposed Methodology

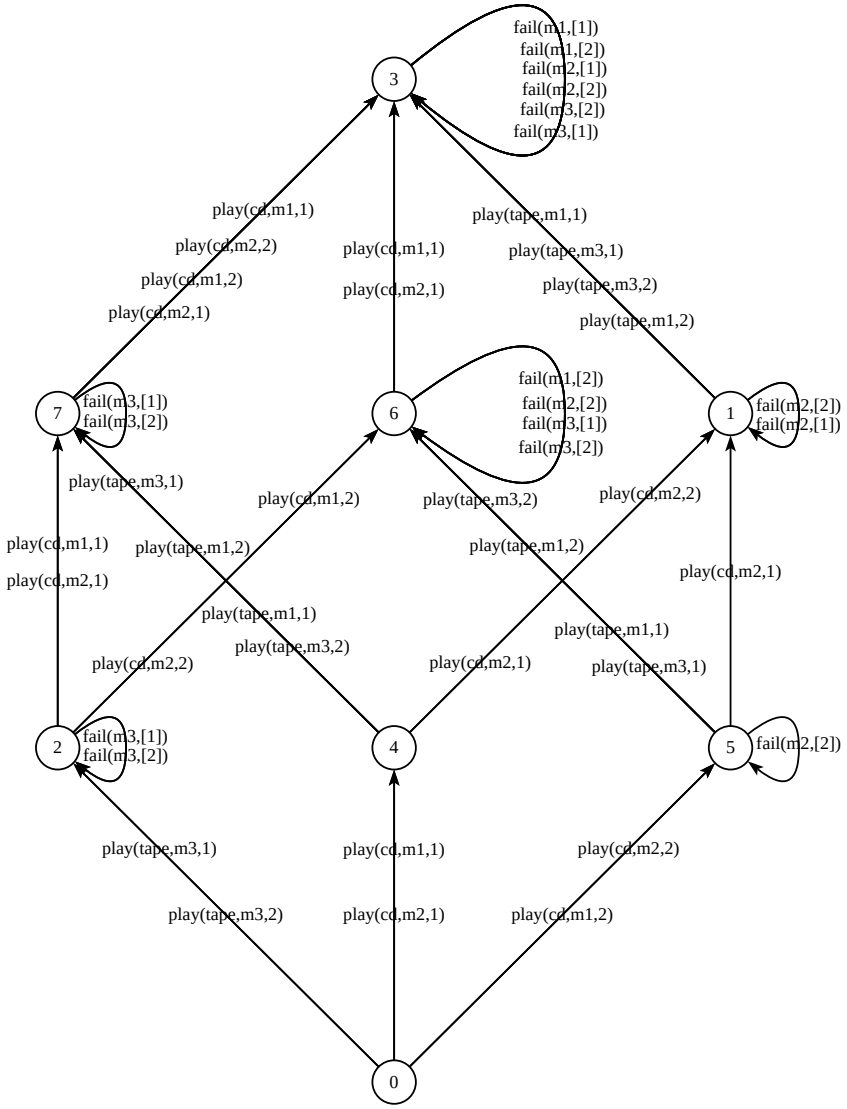


- A high level GUI separates the theoretical details from the users of the methodology
- Non determinism is introduced for sumulating the users
- Changes are performed in μCRL in order to optimize the state space

The Tool


ui_main
VodkaV Models List
Models
NAME FOLDER CREATED MODIFIED VERSION
Design1 design1 May, 2002 May, 2002 simple
Add Del Edit Go
VodkaV Checking Properties
General formulae
SHORTEST FAIL View Check Diagnostic
LONGEST GEN SUCCESS View Check Diagnostic
MAX NUM PLAYS View Check Diagnostic
MAX NUM PLAYS FOR View Check Diagnostic
Scenario analysis
After: play(m1) play(m1)
Can happen: play(m2)
View Check Diagnostic
Scenario analysis
Property: [true*]['fail.*']false
View Check Diagnostic
Save Exit
VodkaV Model Data
NAME TYPE MAX_CON MAX_BW NEXT DESCRIPTION
streaming streaming_level 10 2000 50 storage
storage_group storage_group 10 2000 70 devices
Add Del Edit
NAME TYPE MAX_CON MAX_BW COST MOs
storage1 tape_storage 10 1000 150 m1 m4 m6 m7
storage2 disk_storage 10 2000 80 m2 m4 m5 m7
storage2 cd_storage 10 3000 100 m3 m5 m6 m7
Add Del Edit
Draw supTree Generate model View model Check props Save Exit
VodkaV Storage Edit
Name storage2
Type disk_storage
Max. connections 10
Max. bandwidth 2000
Cost 80
m7 m4 m1 m5
->
Del
m2 m4 m5 m7
Modify Cancel
VodkaV Level Ed
Name
Type
Max. connections 10
Max. bandwidth 2000
Cost 50
Next storage
Modify Cancel

Example of a Simple Reduced Graph

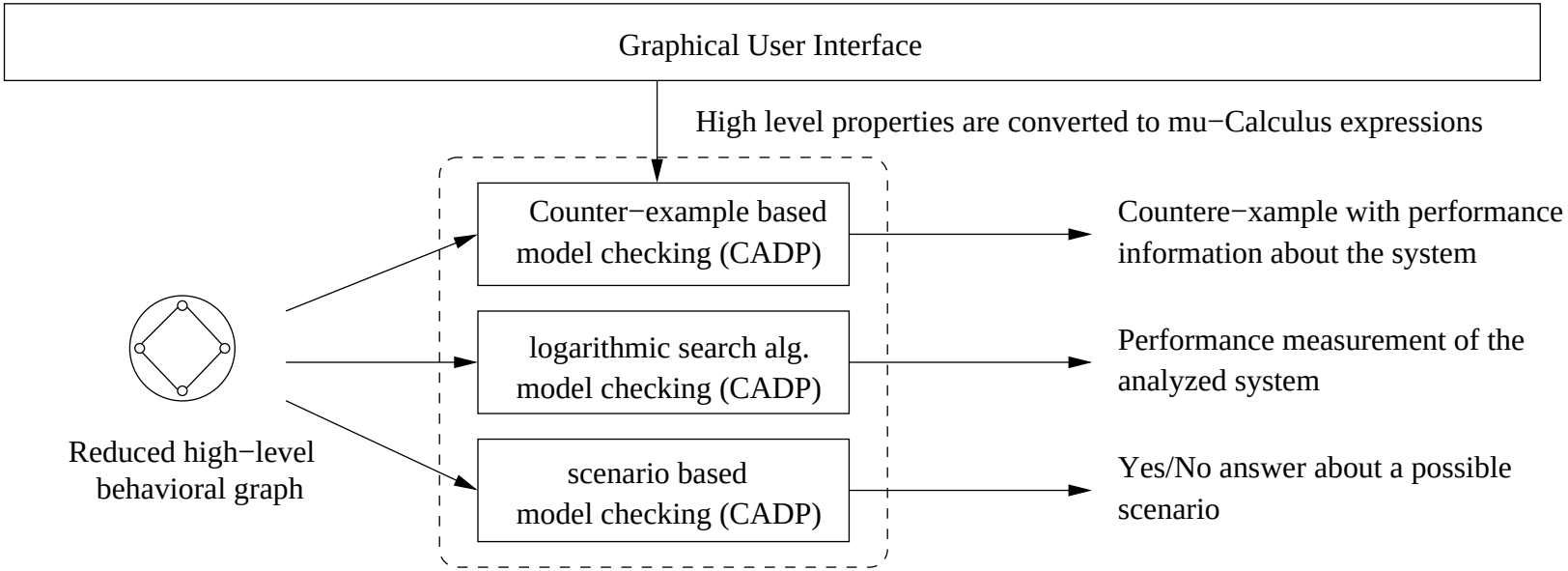


- Two linear levels: streaming level and massive storage level
- Two storage devices:
  - Tape with 20MBit/s, no simultaneous access
  - CD with 30MBit/s, 2 simultaneous access
- No extra restrictions than the trivial cost functions
- Abstract approach for the MOs (m1 in both, m2 and m3 in one of them)
- Two possible qualities: 10/20 MBits/s
- Original state space of 2547 states and 2747 transitions, the reduction results in the 8 states and 48 transitions

Step Three: Extracting Performance Information

- Verifying global properties about the capacity of the system
- Extracting bottleneck information
- Extracting the system architecture

Step Three (I): Verifying Global Properties (black-box)



- 'worst case scenario in which the system reaches its maximum load': `[true*]<not 'fail.*'>true`

- 'maximum number of simultaneous users after which a next user always can be served': `[true*](<'fail.*'>true\/<true>['fail.*']false)`

Step Three (II): Bottleneck information

- **User observational bottleneck**: 'the point in the architecture that makes the first fails to be answered to a user request'.

  Graph analysis of the fails in the top level

- **Internal bottleneck**:
  - 'First place were we can see a fail in the system, in any of the possible execution paths'.

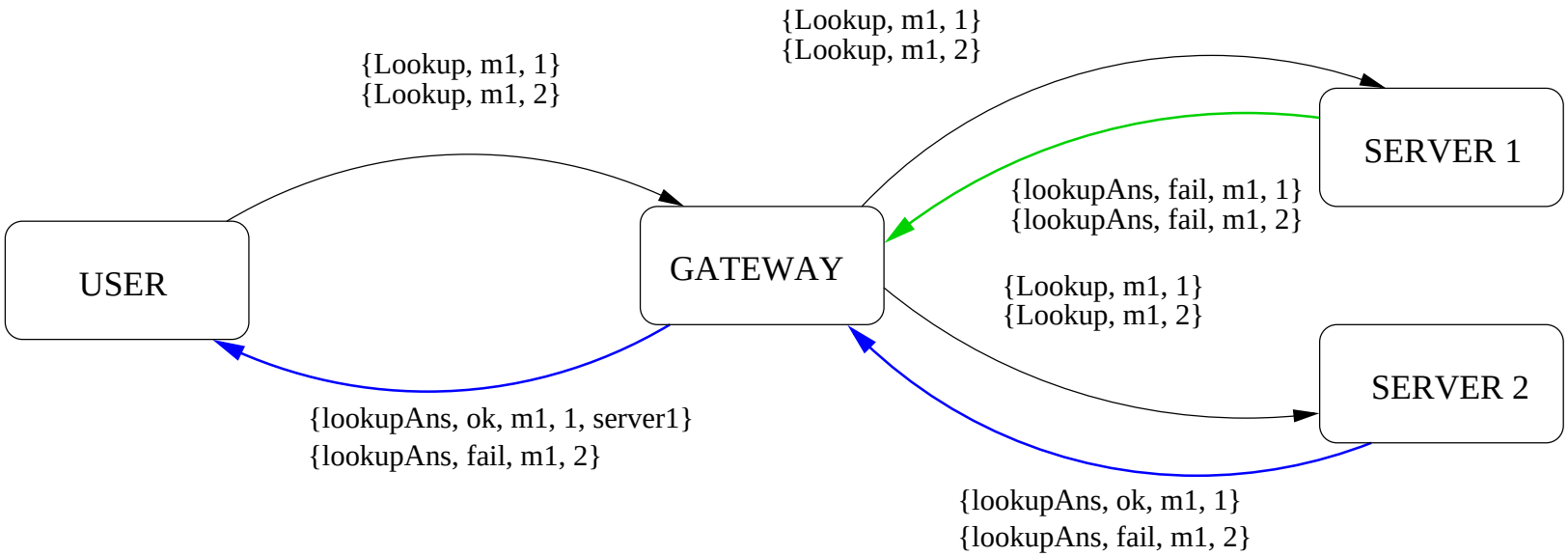
    Stopping the graph generation when a fail occurs
  - 'The part of the system where a fail in a component is too far away from a fail in a different component'.

    Model checking with formulae talking about the distance between fails

For all of them, using graph analysis tools, we can extract the table summary with statistic information about the bottlenecks in the system.

Step Three (III): Architecture from the Messages

- Can we extract the process/component architecture and the protocol of the messages they exchange, from the analysis of the source code?
- In the generic servers: source process, destination process, and message are easy to extract from the analysis of the code
- We can build graphs of this shape:



Conclusions

- **We can get performance information from the source code**
  - Case study: VoDKA, a distributed functional VoD server
  - We use formal methods techniques for extracting information
- **We use the fact that the systems are built on top of OTP** modules and design principles in order to be able to handle complex systems with model checking
- **The methodology can be used in other distributed systems**
- **Some advantages** over testing, tracing and simulation
- **Theoretical details hidden** to the user (designer, programmer)