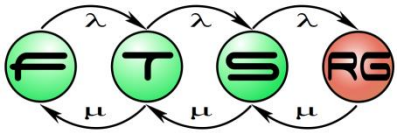
Case Study:
Testing a Complex Algorithm Framework

Attila Klenik

Budapest University of Technology and Economics
Fault Tolerant Systems Research Group



FRAMEWORK OVERVIEW

PetriDotNet

- Developed at BME-MIT, FTSRG
- Tool for formal modeling and analysis
  - For education: Formal Methods course
  - For research: industrial applications
  - .NET based
- Qualitative analyses
  - Model checking
- Quantitative analyses
  - Stochastic analysis

Quantitative Analysis

System plan

Engineering measures

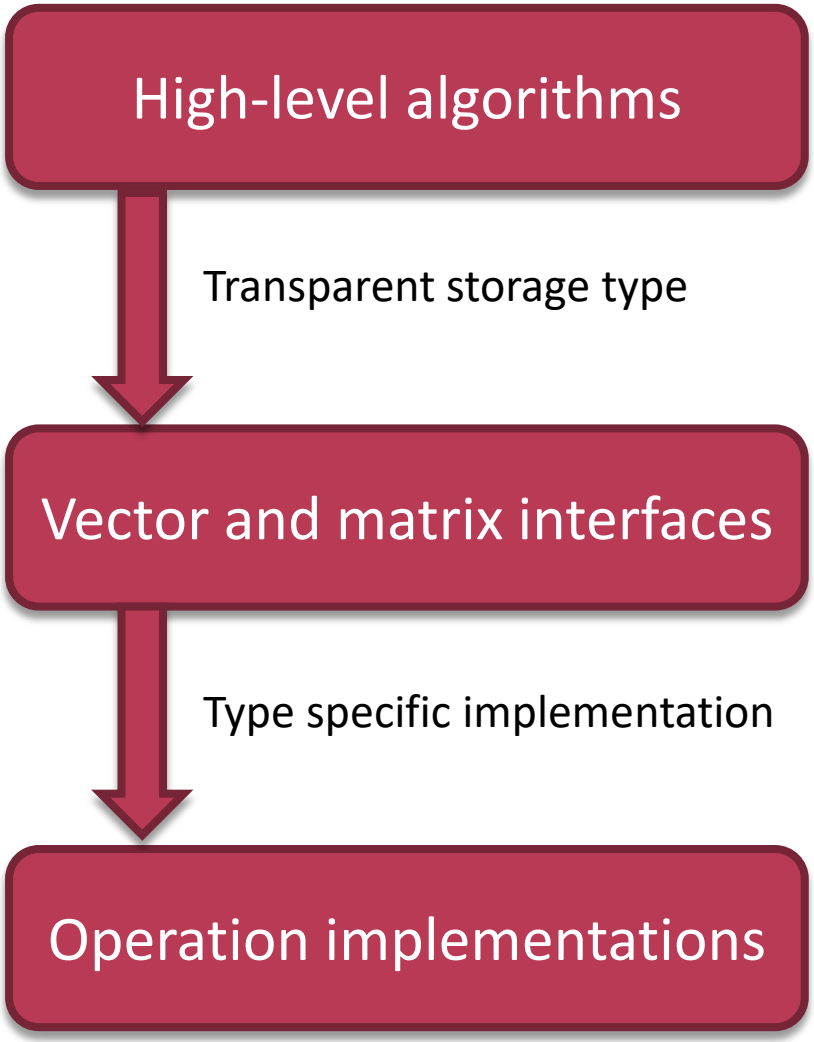
Formal probabilistic model

Formal performance metrics

Stochastic analysis

e.g **Performance** and **Reliability** measures
→ Is the specification satisfied?

Backend Architecture



- Don’t mind the actual data type
- Use it like in pseudo-code

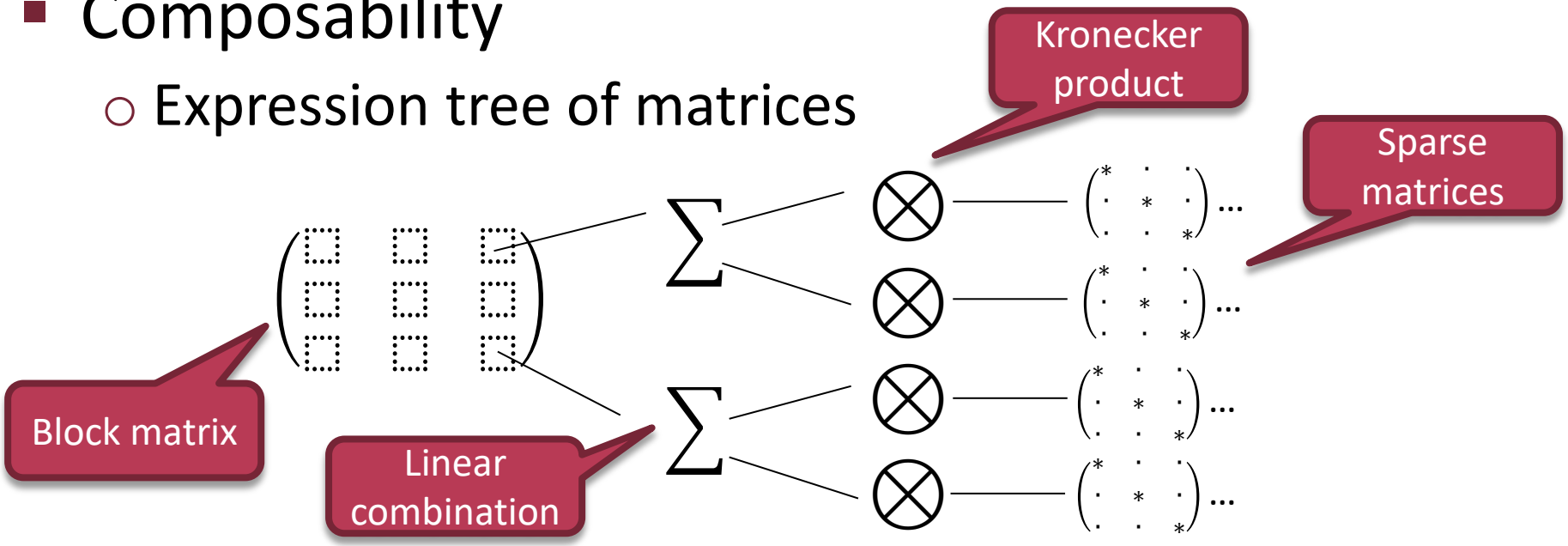
- Provide simple operations
- Hide storage scheme

- Efficient implementations based on storage scheme

TESTING THE VECTOR AND MATRIX SUBSYSTEM

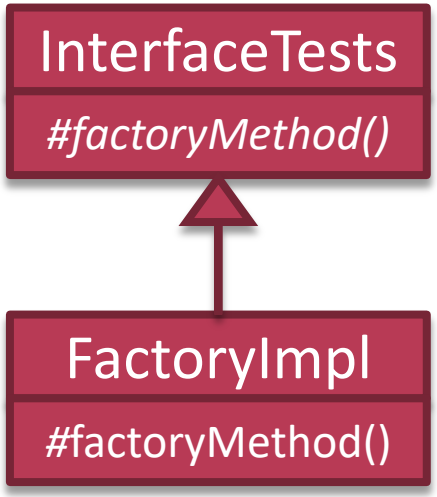
Motivation

- Yet another math lib?
- Not really
  - Simple operations
    - No inversion, spectral decompostion, etc.
  - Efficient implementations
- Composability
  - Expression tree of matrices



Interface Testing - Prerequisites

- Well defined interface for vectors and matrices
- Well defined functional requirements
  - Mathematical correctness ☺
  - Independently of storage scheme
- Interface testing methodology



- Write tests for the interface methods („*abstract*” tests)
- Leave the object instantiation to derived classes („*concrete*” tests)

.NET Utilities

- CodeContracts
  - Preconditions, postconditions, invariants for methods
  - Definition on interface level
    - Every implementing class will have the contracts
- IntelliTest (Pex)
  - Generating tests to cover code blocks (SMT solver)
  - White box testing
- Putting it together
  - Contracts cover multiple errors in one check
  - Pex generates inputs for one check
  - Need to flatten the error checking codes
    - We want test inputs for every error possibility

Creating „Abstract” Tests

- Testing for errors
  - Contract throws exception on every error
    - Can’t cover entire contract with one erroneous input
    - Pex generates multiple inputs to cover every (error) block
  - Use the inputs generated by Pex
- Testing with valid inputs
  - Majority of inputs are indices and sizes
  - Use interval testing around crutial values
- Expected results
  - Calculate it with by-the-book, simple algorithms
    - Simple form of software rendundancy

Manual Vs. Automatic Testing

- Operation: split a vector into blocks of vectors
- Input: sizes of required blocks (long[] parameter)
- Req.: sum of block sizes equals original size
- Most surprising generated test input
  - `long[] blockSizes = {2252471894865346563L, 7150660642841915409L, 4215399814742933512L, 4828211721259356192L}`
  - Sums to 60, perfectly valid inputs ☺
  - We rarely test for overflow and underflow errors

Creating „Concrete” Tests

- Operation definitions follow a certain structure
  - Base: the object we call the operation on
  - Operand: the operand for the operation
    - E.g. Base += Operand
  - Target: the target storage for the operation result
    - E.g. Target = Base + Operand
- Each is behind a general interface
  - No re[illegible]fact[illegible]sed by
  - No re[illegible]ictions (yet) for [illegible]e underlying [illegible]orages
  - E.g. BlockVector = ConstVector + RowVector
  - 8 vector types, 8 matrix types →many combinations

Multiple smaller vectors

Every element is the same

References a row of a matrix

Combinatorial Testing I.

- „Create” derived test classes for every type combination
- Manually not tractable
  - Hundreds of type combinations
  - What if we add a new type
- MS Text Template Transformation Toolkit (T4)
  - Kind of like PHP template engines
  - Generate code with an executable template
  - Generated classes inherit the test methods
  - Possible to override expected behavior ☺

Combinatorial Testing II.

- Type combination generation
  - Automated Combinatorial Testing for Software (ACTS)
  - Capable of n-wise parameter generation
- Full combinatorial testing
  - More than 70k test methods
  - In case of breaking changes, or extra added types
- Pair-wise testing
  - Around 10k test methods ☺
  - In case of simple implementation changes

Importance of Tests

- What we have:
  - Hundrends of „abstract” tests (scenarios)
  - Hundreds of type combinations (with „concrete” tests)
  - Custom behaviors for „concrete” tests
- What if a test fails?
  - Bug: correct it
  - Can’t find implementation for type combination
    - Impl. exists, but delegation is wrong: correct it
    - No implementation
      - Type combination should be supported: implement it
      - Can’t support it (impossible, or not efficient): add custom behavior
- We have a de-facto specification for the subsystem

TESTING THE ALGORITHM SUBSYSTEM

Quantitative Analysis

System plan

Engineering measures

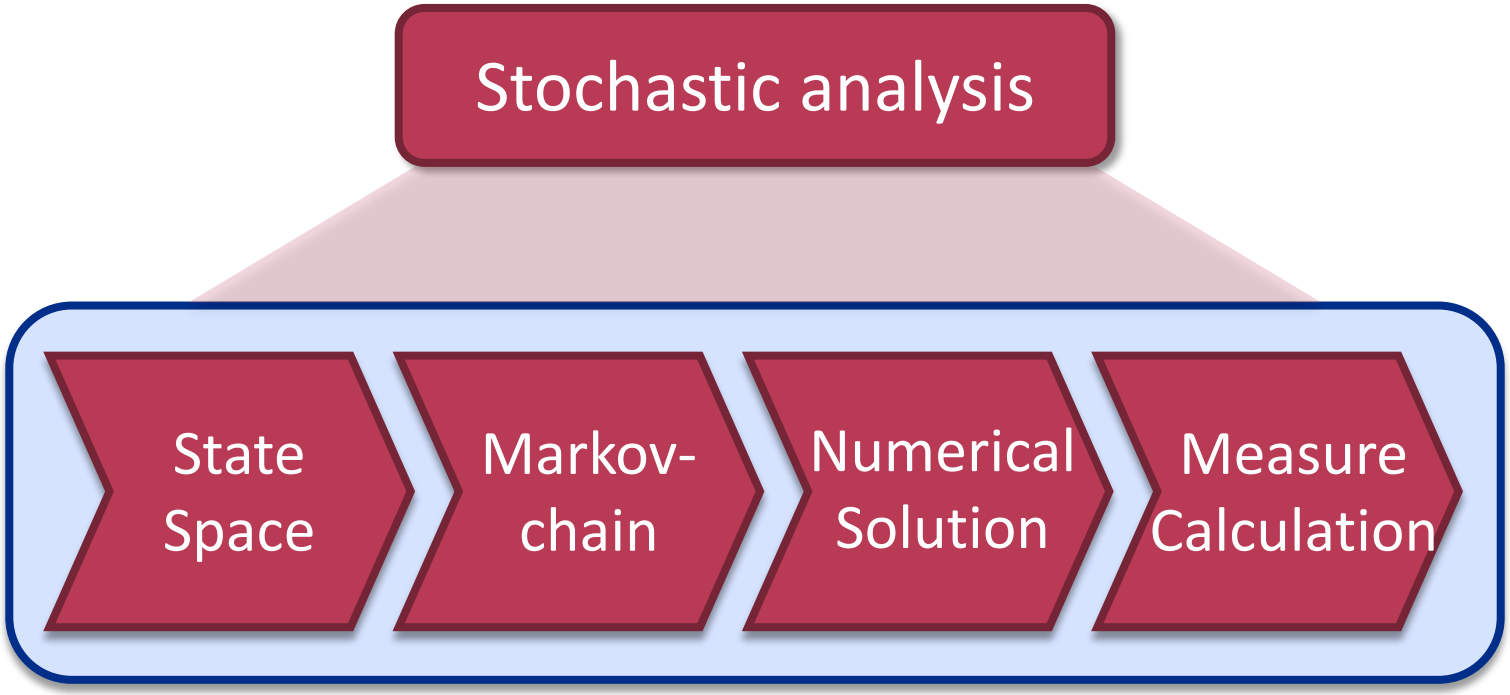
Formal probabilistic model

Formal performance metrics

Stochastic analysis

e.g **Performance** and **Reliability** measures
→ Is the specification satisfied?

Stochastic Analysis Workflow



- Multiple algorithm choices for each step (configurable)
- Complex, hard-to-debug algorithms
- Hard to isolate them
  - The (non-trivial) output of one is the input of an other

Software Redundancy-based Testing

- Inputs of the workflow
  - Formal model
  - Formal measure definitions
- The output is „mathematically given”
  - Even if we don’t know what it is
  - E.g. unique solution of a system of linear equations
- Independent of the used algorithms
  - Run the workflow with different configurations (~500)
  - Compare the calculated metrics
  - Should be the same with respect to a certain numerical precision

Summary

- Vector and matrix library testing
  - Interface testing
  - Automatic test input generation (Pex)
  - Interval testing
  - Combinatorial testing (ACTS)
  - Code generation (T4)

- Algorithm workflow testing
  - Software redundancy