High(er) Assurance Blockchains: Functional and Performance Verification in the Software Design Process

Dr. Neil Davies, Chief Scientist
Predictable Network Solutions Ltd
Cardano

Globally distributed blockchain ledger
Distributed ledger

Participating nodes keep copies of the ledger

- organised as a sequence of blocks containing ‘transactions’

- blocks linked by cryptographic checksums

- authenticity of a block is assured by the sequence of subsequent blocks that refer back to it (the ‘blockchain’)

Consensus algorithm

Consensus on ‘correct history’ achieved by ‘proof of stake’ algorithm [1]

- much more efficient than Bitcoin (‘proof of work’)

- ‘stake’ measured with a value token (cryptocurrency), called ‘Ada’

- currently worth c. $5.4Bn

Initial development funded by IOHK
Cardano – particular challenges

• Blocks and transactions must be diffused to many participating nodes in 20s
  • Nodes can be anywhere in the world
  • Topology is random (to mitigate certain attacks) and dynamic (nodes can join and leave at will)

• Some nodes are assumed to be potentially adversarial
  • Anyone can put up a node and join the network

• All decision-making must be distributed
  • No central point of management

• The system should have a high level of assurance
  • $Billions of value are at stake
Verification goals

Functional verification

• Ensure the concrete implementation does not invalidate assumptions of published correctness proof [1]
• Ensure correct ‘normal’ operation
• Prevent/mitigate attacks and exploits:
  • Against individual users
  • Against the system as a whole

Performance verification

• Mitigate attacks and exploits against the system as a whole
• Ensure correct operation as the system scales
• Ensure individual users receive acceptable service

See: https://iohk.io/research/library/
### Verification issues

<table>
<thead>
<tr>
<th>Issue</th>
<th>End users</th>
<th>Overall system</th>
</tr>
</thead>
<tbody>
<tr>
<td>Align intentional / denotational / operational semantics</td>
<td>Formalising behaviour of ‘wallets’</td>
<td>Formalising operation of consensus algorithm</td>
</tr>
<tr>
<td>Scaling in time and space (computational / state complexity)</td>
<td>Ensuring correct and performant operation as transaction rate increases, and for corporate (exchange) vs consumer use</td>
<td>Maintaining performance as number of participating nodes / users / transactions increases</td>
</tr>
<tr>
<td>Robustness against adversaries</td>
<td>Ensuring privacy and anonymity</td>
<td>Resisting DoS attacks, network partitions, node failures etc.</td>
</tr>
</tbody>
</table>
## Verification activities

<table>
<thead>
<tr>
<th>Issue</th>
<th>End users</th>
<th>Overall system</th>
</tr>
</thead>
</table>
Semi-formal high-assurance approach

Using ‘simulation properties’
1. Write a **formal** specification  
   a. capture *intentional semantics*  
   b. define *invariants*

2. Translate into Haskell **executable** specification  
   a. using referentially-transparent object-oriented approach [8]  
      - ‘smoothing over’ real-world intricacies  
   b. translate model *and* invariants

3. Test (translated) invariants  
   a. using Haskell ‘QuickCheck’ to perform property-based tests on the executable specification  
      - easier than formal proof!  
   b. check consistency  
      - or produce counterexample
Exploiting executable specifications (2)

- Refine executable specification by adding real-world details
  - different computational complexity structures, I/O, message-passing, optimisations, etc.
  - can do this in several stages
  - ensuring there is an ‘abstraction’ relation back to the specification
- Use the executable specification as an ‘oracle’ for verification
  - simulation property
Performance specification and verification
Quantifying performance risks

Support stages of the SDLC

• Design
  • Feasibility analysis
  • Hierarchical decomposition
  • Subsystem acceptance criteria

• Verification
  • Checking delivery of quantified outcomes
  • Evaluating resource usage
  • Re-verification during system lifetime

• Validation
  • Quantification of performance criteria
  • Checking coverage and consistency

Quantify hazards

• Failure to meet outcome requirements
  • Physical constraints
  • Schedulability constraints
  • Supply chain constraints

• Failure to meet resource constraints
  • Scaling
  • Correlations
Nature of performance

• In an ‘ideal world’, systems would always respond instantaneously
  • and without exceptions/failures/errors
• In practice this doesn’t happen
  • there is always some delay and some chance of failure: some impairment
• Thus performance is a privation
  • the absence of impairment
  • like ‘darkness’ or ‘silence’
• Quantity also matters
  • require a certain rate or volume of responses with a given bound on impairment
Capturing sources of impairment

Process algebra

Causality

Information:

- Computed
- Communicated
  * Takes time!

Synchronisation

Explicit

- Communication
- Data dependency

Implicit: resource sharing

- Exclusive
  * Discrete: Locks etc
  * Long timescales
- Statistical
  * Continuous: CPU, interface...
  * Short timescales

Imperfection

Discrete

- Exceptions
- Failures

Statistical

- Resource exhaustion

Stochastic process algebra

Quality Impairment Framework

process algebra

Causality

Information:

- Computed
- Communicated
  * Takes time!

Synchronisation

Explicit

- Communication
- Data dependency

Implicit: resource sharing

- Exclusive
  * Discrete: Locks etc
  * Long timescales
- Statistical
  * Continuous: CPU, interface...
  * Short timescales

Imperfection

Discrete

- Exceptions
- Failures

Statistical

- Resource exhaustion

© Predictable Network Solutions Ltd 2018
Measure of performance: $\Delta Q$

- $\Delta Q$ is a **measure** of the ‘quality impairment’ of an outcome
  - The extent of deviation from ‘instantaneous and infallible’
  - Nothing in the real world is perfect, so $\Delta Q$ **always exists**
- $\Delta Q$ is **conserved**
  - A delayed outcome can’t be ‘undelayed’
  - A failed outcome can’t be ‘unfailed’
- $\Delta Q$ can be **traded**
  - E.g. accept more delay in return for more certainty of completion
- $\Delta Q$ has an **algebra**
  - Can manipulate it mathematically
Performance specifications

- Specify performance in terms of *bounded impairment* ($\Delta Q$)
  - also bounded resource consumption
- Embed this in the formal model ($\chi$-calculus)
  - captures high-level process relations
- Derive dynamic property tests
  - properties over traces - in particular timed traces
1. Decompose the performance requirement following system structure
   • Using engineering judgment/best practice/cosmic constraints
   • Creates initial subsystem requirements
2. Validate the decomposition by re-combining via the behaviour
   • Formally and automatically checkable
     • Can be part of continuous integration
   • Captures interactions and couplings

• Necessary and sufficient:
  • IF all subsystems function correctly and integrate properly
  • AND all subsystems satisfy their performance/resource requirements
  • THEN the overall system will meet its requirements
• Apply this hierarchically until
  • Behaviour is trivially provable OR
  • Have a complete set of testable subsystem verification/acceptance criteria
References

Backup material
Representations of $\Delta Q$

$\Delta Q$ can be represented with an improper random variable

- Combines continuous and discrete probabilities
- Thus encompasses normal behaviour and exceptions/failures in one model

$\Delta Q$ is composable

- Supports hierarchical V&V
Quantifying intent

• The key challenge is to establish *quantified intentions*
  • For outcomes/resources/costs

• Then a variety of mathematical techniques can be applied
  • queuing theory
  • large deviation theory
  • $\Delta Q$ algebra

• This is “only rocket science”
  • Not brain surgery!
  • Set of tools already developed
Quantifying timeliness

**Outcome requirement:**

Suppose we have a specification for how long it’s acceptable to wait for a system outcome:

- 50% of responses within 5 seconds
- 95% of responses within 10 seconds
- 99.9% of responses within 15s
- 0.1% failure rate

This can be represented by an improper CDF.
Meeting a timeliness requirement

Suppose the black line shows the delivered \( \Delta Q \)

- From measurement, simulation or analysis

This is everywhere above and to the left of the requirement curve

- This means that the timeliness / residual failure requirement is satisfied
- If not, there is a performance hazard
Resources
Types of resources

Ephemeral
- Examples:
  - CPU cycles
  - Communication capacity
- Consumption increases $\Delta Q$
  - Removes opportunity to access resources immediately
  - More consumption removes more opportunities, increases $\Delta Q$ further

Threshold
- Examples:
  - Memory
    - Including cache etc.
  - Stored energy
- Consumption only increases $\Delta Q$ when the threshold is reached
  - Then a large increase!
Starting with a functional decomposition:

- Take each subsystem in isolation
  - analyse performance
    - modelling remainder of system as ΔQ
  - quantify resource consumption
    - may be dependent on the ΔQ
- Examine resource sharing
  - within system – quantify resource costs
  - between systems – quantify opportunity cost
- Successive refinements
  - consider couplings
  - iterate to fixed point
Outline of the methodology
Capture concurrency and dependencies

Distinct steps to achieve the outcome

Serial/parallel flow-graph (SPFG)
Model performance

Minimum time to complete action

Time to complete each step
Model resource consumption

Resource consumed to complete each step

Minimum resources to complete action
Apply hierarchically

‘Shape’ of block depends on internal scheduling