Mind Your Outcomes

The ΔQ approach to Quality-Centric Systems Development and Its Application to a Blockchain Case-Study

Seyed Hossein HAERI [1,2] , **Peter Thompson** [3] , **Neil Davies** [3] , **Peter Van Roy** [4] , **Kevin Hammond**[5] , **James Chapman**[5]

[1]   Formal Methods Team, IOHK; hossein.haeri@iohk.io
[2]   Department of Informatics, University of Bergen, Norway; hossein@uib.no
[3]   Predictable Network Solutions Limited, Stonehouse, U.K.; [Peter.Thompson,Neil.Davies]@pnsol.com
[4]   Université catholique de Louvain, Belgium; pvr@info.ucl.ac.be
[5]   IO Global; [kevin.hammond,james.chapman]@iohk.io
*   Correspondence: hossein.haeri@iohk.io
†   This paper is an extended version of our paper published in 33rd Symposium on Implementation and Application of Functional Languages (IFL21), as a draft.

Abstract: This paper directly addresses a critical issue that affects the development of many complex distributed software systems: how to establish quickly, cheaply and reliably whether they will deliver their intended performance *before* expending significant time, effort and money on detailed design and implementation. We describe ΔQSD, a novel *metrics-based* and *quality-centric* paradigm that uses formalised *outcome diagrams* to explore the performance consequences of design decisions, as a performance blueprint of the system. The ΔQSD paradigm is both effective and generic: it allows values from various sources to be combined in a rigorous way, so that approximate results can be obtained quickly and subsequently refined. ΔQSD has been successfully used by Predictable Network Solutions for consultancy on large-scale applications in a number of industries, including telecommunications, avionics, and space and defence, resulting in cumulative savings of $Bs. The paper outlines the ΔQSD paradigm, describes its formal underpinnings, and illustrates its use via a topical real-world example taken from the blockchain/cryptocurrency domain, where application of this approach enabled an advanced distributed proof-of-stake system to meet challenging throughput targets.

Keywords: formal semantics; quality attenuation; distributed systems; system design; scalability; performance; feasibility; blockchain; ΔQ.

---

1. Introduction

The modern world depends on large-scale software systems to run its critical infrastructure: telecommunications, energy, finance, transport, government, the military, and major multi-national companies all rely on correct and reliable software to carry out their day-to-day operations. Individuals in many countries increasingly rely on online systems for many aspects of their daily lives. Such systems are complex and expensive to construct and maintain. All too often, however, they fail to deliver the intended performance, or sometimes even to meet the most basic requirements of reliability and usability. One example of the economic impact of failing to adequately manage performance is OnLive. OnLive was a cloud gaming platform that attempted to deliver high quality video and real-time interactivity, but failed to deliver an acceptable experience [1]. As a consequence, the company folded, wiping out a $1.8B valuation and costing investors several hundred million US dollars [2,3].

Such results are, fundamentally, a failure both of design and of management. This has a major social and economic cost: fixing a problem in development typically costs 10 times as much as fixing it in design, and fixing a problem in a released product typically costs 100 times as much as fixing it in development [4]. Overall, the cost of failed software projects in the US in 2020 was approximately $260B (up from $177.5B in 2018) [5].

One core cause of problems such as these is that modern software development practices successfully emphasise rapid and flexible software construction, but fail to adequately consider essential quality requirements, or even to consider properly whether a system can actually meet its intended outcomes, particularly when deployed at scale. In complex system development there is a tendency for cost/performance hazards to appear late in the System Development Life Cycle (SDLC). Unfortunately, such issues can invalidate design choices, requiring major redesign or implementation changes. This can lead to time and cost overruns, and sometimes project cancellation. Thus there is an urgent need for verification and validation of resource cost and system performance (as opposed to simply functional correctness), and for this to be part of an ongoing design process rather than applied late in the development. This process must be applicable to distributed and complex hierarchical systems, and support both initial and incremental development. This methodology becomes increasingly important as such systems are used for cyber-physical, mission-critical or even safety-of-life applications. Several factors make this task more difficult:

1. System requirements are often vague and/or contradictory, and can change both during and after development;
2. Complexity forces hierarchical decomposition of the problem, creating boundaries , including commercial boundaries with third-party suppliers, that may hinder optimal development and obscure risks;
3. Time pressure forces parallel development that may be at odds with that hierarchical decomposition, and encourages leaving 'tricky' issues for later, when they tend to cause re-work and overruns, and leave tail-risks;
4. Cost and resource constraints force resources to be shared, both within the system and with other systems (e.g., when network infrastructure or computing resources are shared); they may also require re-use of existing assets (own or third-party), introducing a degree of variability in the delivered performance;
5. The performance of particular components or subsystems may be incompletely quantified;
6. System performance and resource consumption may not scale linearly (which may not become apparent until moving from a lab/pilot phase to a wider deployment);
7. At scale, exceptional events (transient communications and/or hardware issues) can no longer be treated as negligibly rare and their effects and mitigation need to be considered, along with the associated performance impacts.

Thus, what is needed is: (1) a way of capturing performance and resource requirements that accommodates *all* the various sources of uncertainty; and, (2) a process for decomposing a top-level requirement into subsystem requirements that provides confidence that satisfying all the lower-level requirements will also satisfy the top-level one. For functional aspects of system behaviour there are various ways of dealing with this [6]. In terms of performance – which for time-critical systems design means delivering an outcome within a time bound (not just at a particular rate) – there are established approaches [7], but these all have significant limitations.

This paper directly addresses those issues by defining the ΔQSD systems development paradigm and providing a high-level formalism that can be used throughout the software development process. It illustrates the use of ΔQSD on an actual real-time distributed system, namely the Cardano blockchain diffusion algorithm. ΔQSD is a quality-centric paradigm, focusing on meeting timeliness constraints and acceptable failure rate of the top-level outcomes with acceptable resource consumption. The paradigm has been used successfully by Predictable Network Solutions in many large industrial projects, collectively saving billions of dollars and person-centuries of development effort. It informs

78 high-level management and system design decisions by showing where conflicts exist (or *may exist*)
79 between system designs and required outcomes. ΔQSD presents the design process as a sequence of
80 refinement steps, that can be taken all the way from very high level to a fully specified and constructible
81 system. It is able to compute predicted performance at any stage of the design process, where
82 performance is seen broadly as comprising timeliness, behaviour under load, resource consumption,
83 and other key system metrics. Central to ΔQSD is the concept of an *outcome*, defined as a specific
84 system behaviour with specified start and end points, and localised in space and time. In ΔQSD, the
85 system engineer models the system as an *outcome diagram*, which is a graph that captures the causal
86 relationships between system outcomes.

87 *1.1. Main Contributions of this Paper*

88   The main contributions of this paper are to:

89 1. Introduce ΔQSD, a formalism (Section 5) focused on rapidly exploring the performance
90   consequences of design and implementation choices, where:

91   (a) performance is a first class citizen, ensuring that we can focus on details relevant to
92   performance behaviour;
93   (b) the whole software development process is supported, from checking feasibility of initial
94   requirements to making decisions about subtle implementation choices and potential
95   optimisations;
96   (c) we can measure our choices against desired outcomes for individual users (customer
97   experience);
98   (d) analysis of saturated systems is supported;
99   (e) analysis of failure is supported.

100   We use term-rewriting for formalising refinements (Definition 3 in Section 5) and denotational
101   semantics for formalising timeliness analysis (Section 5.3) as well as load analysis (Section 5.4).
102 2. Describe key decisions made in the development process of a real system – i.e., the Cardano
103   blockchain, presented as a running example – and show how ΔQSD is able to quickly rule out
104   infeasible decisions, predict behaviour, and indicate design headroom (slack) to decision makers,
105   architects, and developers (Section 4).

106 While the ΔQ concept has been described in earlier papers [8] [9], and used to inform a number of
107 large-scale system designs, these previous contributions have only used it in an informal manner. By
108 providing a formal definition of ΔQSD, and showing how it can be used in practice, we are taking an
109 important step towards a general evidence-based engineering methodology for developing real-time
110 distributed systems.
111   As described above, a key contribution of this work is the new formalism of ΔQSD that defines
112 a system design as a sequence of outcome diagrams. This sequence starts with a fully unspecified
113 system and ends either with a fully specified system or a convincingly specified-enough system (both
114 deemed as constructible), or with the conclusion that the system goals are *infeasible*. The formalism
115 allows exploration of the design space by assessing the consequences of the decisions that are taken at
116 each refinement step and possibly retracting, giving rise to threaded decision trees. For each partially
117 specified system, we compute the predicted timeliness and behaviour and resource consumption of the
118 system under load, obtaining one of three possible conclusions: (1) *infeasibility* – hence, ceasing further
119 development and revising former design decisions; (2) *slack* – hence, ceasing further optimisation
120 because the system is good enough; or (3) *indecisiveness* – hence, requiring additional scrutiny until one
121 of the alternative conclusions can be drawn. The paper gives one large example, blockchain diffusion,
122 that illustrates how ΔQSD can be used in practice, explaining how the formalism can be used to drive
123 the design process and associated decision making. This example is a real-world application that is in
124 continuous use as a core part of the Cardano blockchain technology (https://cardano.org/).
125   The requirement that most approaches impose for fully-specified, or even fully-implemented,
126 systems is a serious disadvantage, since it does not allow the properties of subsystems to be

127 encapsulated and hierarchically (de)composed. *Compositionality*, the principle that the meaning of a
128 complex expression is determined by the meanings of its constituent expressions and the rules that
129 are used to combine them, is key to managing complexity within the systems development life-cycle.
130 For compositional properties, what is "true" about subsystems (e.g., their timeliness, their resource
131 consumption) is also "true" about their (appropriate) combination: there exists an invariant (e.g.,
132 timeliness, aspects of functional correctness) that must hold over the reified components of the system.
133 In the broader software development space, functional programming techniques are improving the
134 compositionality of functional aspects of software systems, and can deliver high assurance of functional
135 correctness when combined with appropriate formal methods [10]. The paradigm presented here,
136 which we call ΔQSD, represents a similar step change in handling the "non-functional" aspects of
137 performance and resource consumption. By treating delay and failure as a single object, called 'quality
138 attenuation', our paradigm can be thought of as a combination of passage time analysis and failure
139 mode effects analysis (FMEA).

140 *1.2. Structure of the Paper*

141   This paper has the following structure:

142   1. Section 3 defines the basic concepts that underlie the ΔQSD formalism: outcomes, outcome
143   diagrams, and quality attenuation (ΔQ). We also compare outcome diagrams with more
144   traditional diagrams such as block diagrams.
145   2. Section 4 gives a realistic example of the ΔQSD paradigm, showing a step-by-step design of
146   block diffusion based on quality analysis. (Block diffusion itself is introduced in Section 2.) This
147   example introduces the basic operations of ΔQSD in a tutorial fashion. The example uses realistic
148   system parameters that allow us to compute predicted system behaviour.
149   3. Section 5 gives the formal mathematical definition of ΔQSD and its main properties. With this
150   formal definition, it is possible to validate the computations that are used by ΔQSD as well as to
151   build tools based on ΔQSD.
152   4. Section 6 gives a comprehensive discussion about related work from three different viewpoints:
153   theoretical approaches for performance analysis (Section 6.1), performance design practices
154   in distributed systems (Section 6.2), and programming languages and software engineering
155   (Section 6.3).
156   5. Section 7 summarises our conclusions, and describes our plans to further validate ΔQSD and
157   to build a dedicated toolset for real-time distributed systems design that builds on the ΔQSD
158   paradigm.

2. Running Example: Block Diffusion in the Cardano Blockchain

A blockchain is a form of distributed ledger. It comprises a number of blocks of data, each of which provides a cryptographic witness to the correctness of the preceding blocks, back to some original 'genesis' block (a 'chain' of blocks, hence 'blockchain') [11]. Nodes in the system use some specified protocol to arrive at a distributed consensus as to the correct sequence of blocks, even in the presence of one or more 'adversaries' that aim to convince other nodes that a different sequence is correct. One such consensus protocol is Ouroboros Praos [12], which underpins Cardano (https://www.cardano.org), one of the world's leading cryptocurrencies. Ouroboros Praos uses the distribution of 'stake' in the system (i.e. the value of the cryptocurrency tokens that are controlled by each node) to randomly determine which node (if any) is authorised to produce a new block in the chain during a specific time interval (a 'slot'); the more stake a node controls, the more likely it is to be authorised to produce a block. For this to be effective, it is important that the block-producing node has a copy of the most recently produced block, so that the new block can correctly extend the existing chain. Since the block producer is selected at random, this means that the previous block needs to have been copied to *all* block-producing nodes; we call this process 'block diffusion'. Since blocks are produced on a predetermined schedule and each block depends on its predecessor, block diffusion is a hard real-time problem; each block must be diffused before the next block can be produced. In order to be robust, however, the consensus algorithm is designed to withstand some imperfections in block diffusion, hence the effective requirement is that blocks should be well-diffused "sufficiently often". Put another way, the probability that a block fails to arrive in time for the production of the next block must be suitably bounded. The engineering challenge is to quantify this probability as a function of the design and of the parameter choices of the implementation.

The scale of the challenge is illustrated by Cardano. Cardano is a global-scale distributed system that eschews centralised management. At the time of writing, 2,948 globally-distributed nodes cooperate to produce and distribute blocks for $45.77B of cryptocurrency that is associated with 956,092 distinct user addresses. The stake distribution at the time of writing is shown in Fig. 1.

In Cardano, slots are one second long and blocks are produced every 20 seconds on average. An initial implementation of Cardano (code-named 'Byron') was functionally correct but proved incapable

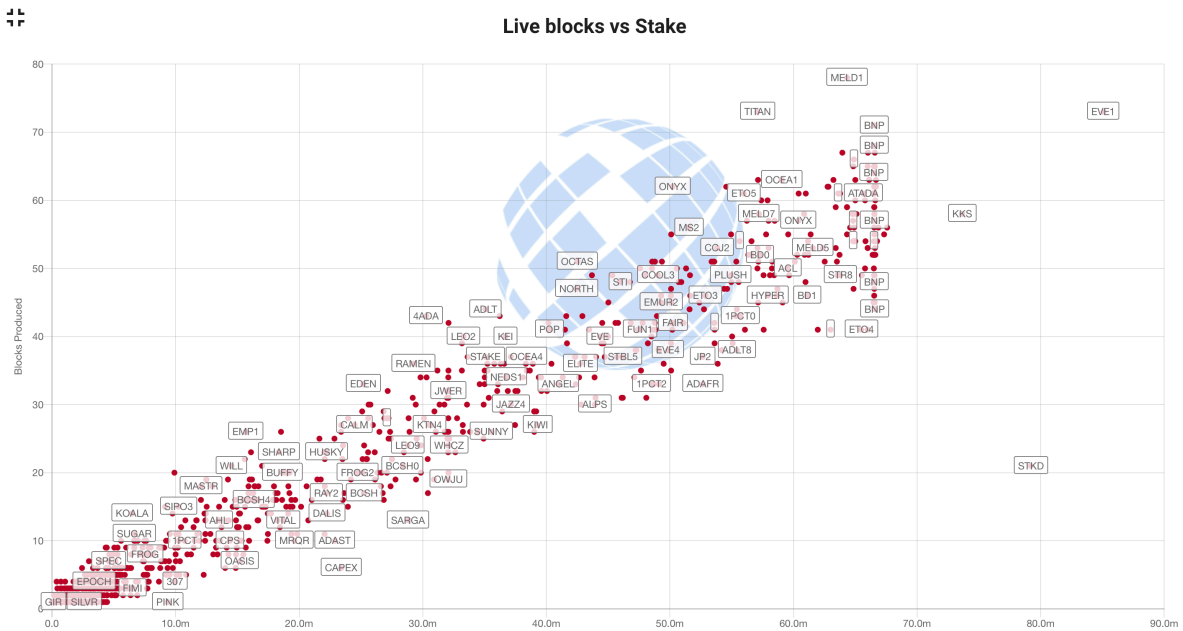



Figure 1. Distribution of Stake by Node on the Cardano Blockchain, November 2021 (source: https://pooltool.io/analysis). The y-axis represents the number of blocks produced by each "stake pool"; the x-axis represents the stake that is held by the pool (in Ada).

187 of effective block diffusion without rigid control of the nodes and their topology; a re-implementation
188 (called 'Jormungandr') targeted higher performance by using a different programming language
189 (Rust instead of Haskell), but this also missed the block diffusion target by a wide margin. A further,
190 and ultimately successful, re-implementation (called 'Shelley' [13]), used Haskell to retain strong
191 correctness assurances, but applied the principles that are discussed in this paper to also ensure
192 adequate performance in a fully decentralised deployment.

2.1. Key Design Decisions

194   In the design of Shelley, a number of inter-related decisions had to be made. These included:

195 **How are nodes connected?** It might seem that connecting every node to every other would minimise
196   block diffusion time, however the lack of any control over the number and capabilities of nodes
197   makes this infeasible. Nodes can only be connected to a limited number of peer nodes; the
198   number of connected peers and how they are chosen then become important.
199 **How much data should be in a block?** Increasing the amount of data in a block improves the overall
200   throughput of the system but makes block diffusion slower.
201 **How should blocks be forwarded?** Simply forwarding a new block to all connected nodes would
202   seem to minimise delay, but this wastes resources, since a node may receive the same block
203   from multiple peers. In the extreme case, this represents a potential denial-of-service attack.
204   Splitting a block into a small *header* portion (sufficient for a node to decide whether it is new) and
205   a larger *body* that a node can choose to download if it wishes mitigates this problem but adds an
206   additional step into the forwarding process.
207 **How much time can be spent processing a block?** Validating the contents of a block before
208   forwarding it mitigates adversarial behaviour, but can be computationally intensive, since
209   the contents may be programs that need to be executed (called 'smart contracts'); allowing more
210   time for such processing permits more, and more complex, programs but makes block diffusion
211   slower.

212 The remainder of this paper shows how such design decisions can be quantified using the ΔQSD
213 paradigm.

2.2. Formulating the Problem

215   We assume that a collection of blockchain nodes is assembled into a random graph (randomness
216 is important in a blockchain setting for mitigating certain adversarial behaviours). In each time slot, a
217 randomly-chosen node may generate a block, and we are interested in the probability that the next
218 randomly-chosen node has received that block before it generates the next block.

> **Problem Statement**
>
> Starting from blockchain node A, what is the probability distribution of the time taken for a
> block to reach a different node Z, when A and Z are picked at random from the graph?

219

220 Since the graph is random with some limited node degree N, there is a strong chance that A is
221 not directly connected to Z, and so the block will have to pass through a sequence of intermediate
222 nodes $B, C, \ldots$ The length of this sequence is a function of the size and node degree of the graph [14].
223 The (distribution of) time to forward a block directly from one node to another is known (e.g., by
224 measurement).

3. Foundations

In the remainder of this paper, we will take the *system of discourse* to be fixed for the design engineer. We assume that this system has a number of tasks that must be performed. In order to perform a task that is not considered to be atomic by the design engineer, the system might need to perform several other sub-tasks. The process of clarifying the details of the system by breaking higher-level tasks into such sub-tasks is what we call *refinement* (Definition 3 in Section 5). By refining a system, one goes from a coarser *granularity* of the design to a finer one. (See Sections 4.1–4.3 for examples of that.) Sometimes, the design engineer will return to a coarser grained design (the dashed grey arrows in Fig. 6) in order to take a different direction of refinement. (See Sections 4.4–4.7 for example.) Some reasons why the might want to do that are: to investigate other aspects of their system; to compare two alternative design choices; or because a refinement fails to meet the necessary performance or other requirements. ΔQSD is thus design exploration in the world of refinements.

This section sets the stage for presenting design exploration in action (Section 4) by introducing the fundamental concepts: Outcomes (Section 3.1), Outcome Diagrams (Section 3.2), and Quality Attenuation (Section 3.3). It then, gives a simple example of how to approach problems à la ΔQSD (Section 3.4). This section ends in a discussion on why ΔQSD advises a new diagram in the presence of all the existing ones in Software Engineering (Section 3.5).

3.1. Outcomes

An *outcome* is what the system obtains by performing one of its tasks. Each task has precisely one corresponding outcome and each outcome has precisely one corresponding task. We say that an outcome is 'performed' to mean that the corresponding task of an outcome is performed. Likewise, we might use task adjectives for outcomes too, even though outcomes and tasks are inherently different. For example, by an *atomic outcome*, we mean an outcome whose corresponding task is itself atomic.

We take an event-based perspective, in which each outcome has two distinct sets of events: the starting set of events (any one of which must happen before the task can commence) and the terminating set of events (at least one of which must happen before the task can be considered complete). Each of those sets consists of events that are of particular interest (as opposed to just any event). We call such events of interest the *observable*s. For example, an observable in the starting set, S_o, of an outcome o is of interest because it signifies the point in time at which o's performance began as well as the 3D location of o. Collectively, these comprise the 4D location or simply *location* of o. Likewise, an observable from the terminating set, T_o of o is an event that contains information regarding the location of o. While it may seem unusual to refer explicitly to location in a computer science context, when considering distributed systems, the outcomes of interest are precisely those that begin at one location and finish at another. Of course, once an observable from S_o occurs, there is no guarantee that one from T_o will occur within o's *duration limit*, $d(o)$ (i.e., the relative time by which o is required to complete). However, when an observable T_o does occur within the duration limit after one from S_o, o is said to be *done*.

Diagrammatically, we show an outcome using an orange circle. As shown in Figure 2 we depict the starting set and the terminating set of an outcome using small boxes to the left and right of the outcome's circle, respectively. The starting set is connected to the outcome from the left and the terminating set from the right. When they are unimportant for an outcome, we do not include the starting set and the terminating set of that particular outcome in the diagram.

We consider one special kind of outcome. Consider the situation where a design engineer is aware that an outcome is not atomic. They will eventually need to further break the outcome into its sub-outcomes. Nevertheless, the current level of granularity is sufficient to carry out a particular analysis (see Section 5.3 and Section 5.4 for two example analyses). In ΔQSD, a *black box* can be used for that particular outcome. Black boxes are those outcomes that either:

1. Can be easily quantified without even a need for them to be named;

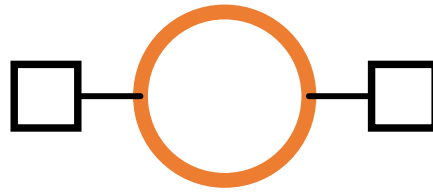



Figure 2. An Outcome with its Starting Set and Terminating Set on its Left and Right

273   2. Are beyond the design engineer's control (and so may need to be quantified by external
274   specification or measurement); or,
275   3. Are ones for which the design engineer has intentionally left the details for later.

276   *Outcome variable*s are the variables that we use to refer to a given outcome.

277   *3.2. Outcome Diagrams and Outcome Expressions*

278   The description of a system in terms of its outcomes requires the causal relationships between
279   the outcomes to be captured. In ΔQSD, these relationships are captured in *outcome diagram*s. In
280   addition to its graphical presentation, each outcome diagram can be presented algebraically, using its
281   corresponding *outcome expression*. As shown in Figure 3, outcome diagrams offer four different ways
282   to describe the relationships between outcomes.

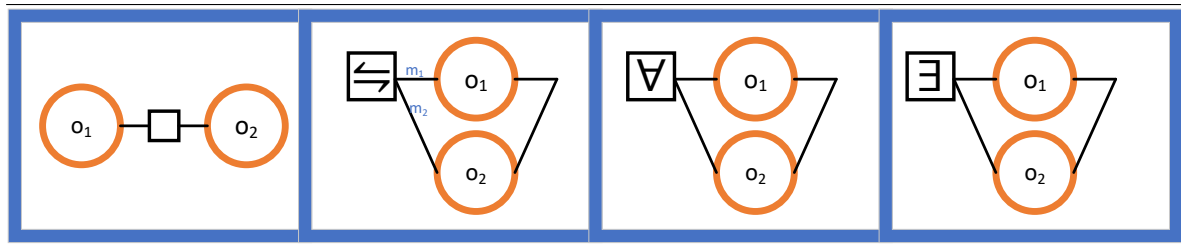



Figure 3. Relationships in an Outcome Diagram. From Left to Right: 1) Sequential Composition; 2)
Probabilistic Choice; 3) All-to-Finish; 4) First-to-Finish.

283   We now explain Figure 3 from left to right.

284   • In the first case, the outcomes o_1 and o_2 are said to be *sequential*ly composed. o_2 therefore causally
285   depends on o_1. We maintain a directional convention to avoid showing directions explicitly:
286   when an edge connects two outcomes, the right one causally depends on the left one. The
287   corresponding outcome expression is "$o_1 \bullet\!\!\rightarrow\!\!\bullet o_2$."
288   • In the second case, a *probabilistic choice* is made between o_1 and o_2. Notice the weights m_1 and
289   m_2. The outcome of the choice is the same as o_1 with probability $\frac{m_1}{m_1+m_2}$ and the same as o_2 with
290   probability $\frac{m_2}{m_1+m_2}$. The corresponding outcome expression is "$o_1 \overset{m_1}{\underset{m_2}{\Longleftarrow}} o_2$."
291   • In the third case, an *all-to-finish* (aka *last-to-finish*) combination is produced from o_1 and o_2. For
292   two outcomes o_1 and o_2 that are started at the same time and that are run in parallel, the outcome
293   is *done* when both o_1 and o_2 are *done*. The corresponding outcome expression is "$\forall(o_1 \parallel o_2)$."
294   • In the final case, a *first-to-finish* combination is produced from o_1 and o_2. For two outcomes o_1
295   and o_2 that are started at the same time and that are run in parallel, the outcome is *done* when
296   either o_1 or o_2 is *done*. The corresponding outcome expression is "$\exists(o_1 \parallel o_2)$."

297   *3.3. Quality Attenuation (ΔQ)*

298   From the perspective of a user, a perfect system would deliver the desired outcome without error,
299   failure or delay, whereas real systems always fall short of this; we can say that the quality of their
300   response is *attenuated* relative to the ideal. We denote this 'quality attenuation' by the symbol ΔQ and
301   reformulate the problem of managing performance as one of *maintaining suitable bounds* on ΔQ [15].
302   This is an important conceptual shift because 'performance' may appear to be something that can be
303   increased arbitrarily, whereas ΔQ (rather like noise) is something that may be minimised but that can
304   never be completely eliminated. Indeed, some aspects of ΔQ, such as the time for signals to propagate
305   between components of a distributed system, cannot be reduced below a certain point.

306   Because the response of the system in any particular instance can depend on a wide range of
307   factors, including the availability of shared resources, we model ΔQ as a random variable. This allows
308   various sources of uncertainty to be captured and modelled, ranging from as-yet-undecided aspects of
309   the design, to resource use by other processes, to behavioural dependence on data values.

310   In capturing the deviation from ideal behaviour, ΔQ incorporates both delay (a continuous
311   random variable) and exceptions/failures (discrete variables). This can be modelled mathematically
312   using *Improper Random Variables* (IRVs), whose total probability is less than one [16]. If we write ΔQ(x)
313   for the probability that an outcome occurs in a time $t \leq x$, then we can define the 'intangible mass' of
314   such an IRV as $1 - \lim_{x \to \infty} \Delta Q(x)$. In ΔQ, this encodes the probability of exception or failure. This
315   is illustrated in Figure 4, which shows the cumulative distribution function (CDF) of an IRV (with
316   arbitrary time units).

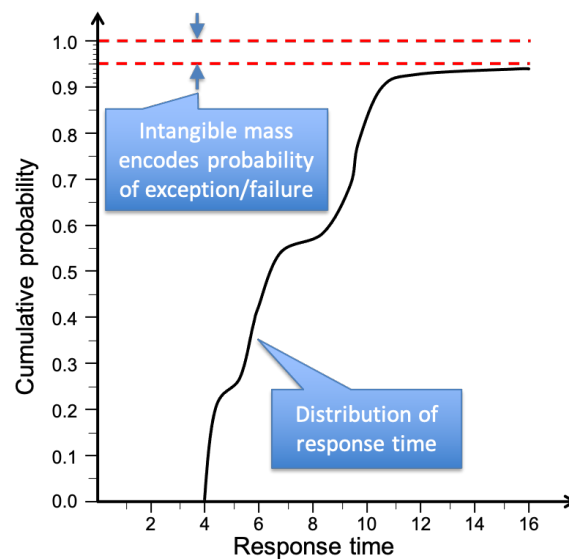



Figure 4. Cumulative Distribution of an Improper Random Variable (IRV).

We can define a partial order on such variables, in which the 'smaller' attenuation is the one that
delivers a higher probability of completing the outcome in any given time:

$$(\forall_x \Delta Q_1(x) \geq \Delta Q_2(x)) \equiv \Delta Q_1 \leq \Delta Q_2 \tag{1}$$

317   This partial order has a 'top' element, which is simply perfect performance: $\top \equiv (\forall_x \Delta Q(x) = 1)$, and
318   a 'bottom' element, which is total failure (an outcome that never occurs): $\bot \equiv (\forall_x \Delta Q(x) = 0)$. We can
319   write specifications for system performance using this partial order by requiring the overall ΔQ to be
320   less than or equal to a predefined bounding case. Where the ΔQ is strictly less than the requirement,
321   we say there is performance *slack*; when it is strictly greater than the requirement, we say there is a
322   performance *hazard*. (Cf. Definitions 5 and 8.)

³²³   Assessments might also find the current level of information about a system to be *indecisive* –
³²⁴ neither slack nor hazard. The simplest reason for indecisiveness is the partiality of ≤ in Equation (1).
³²⁵ Another reason for indecisiveness might be conflict between different analyses. For example, timeliness
³²⁶ analysis (Section 5.3) might show slack whilst load analysis (Section 5.4) shows hazard. A third reason
³²⁷ might be that, even though the formulations end up indicating slack or hazard, the system is still
³²⁸ detailed so little the result of the analysis should not be counted on.

     The relationships between outcomes that were shown in Figure 3 then induce corresponding
relationships between the ΔQs of those outcomes, as explained in Section 5.3. The key to the
compositionality of the approach is that the partial order is preserved by the operations that combine
ΔQs. Thus, for example

$$\Delta Q_1 \le \Delta Q_2 \implies \Delta Q_1 * \Delta Q_3 \le \Delta Q_2 * \Delta Q_3$$

³²⁹ This enables an overall timeliness requirement to be broken into 'budgets' for sub-outcomes. More
³³⁰ details of this approach are given in [17].

³³¹ *3.4. Simple Example*

³³²   Consider the simple distributed system of a web browser interacting with a set of servers that
³³³ collectively provide a web page. The outcome that is of interest to the user starts with the event of
³³⁴ clicking on a URL, and ends with the event of the page being fully rendered. This corresponds to the
³³⁵ first row of Figure 5. The second row shows the distinction between the user and the browser, and
³³⁶ the third row exposes the back-end servers. A typical web page will contain a variety of elements
³³⁷ that are served by servers from different host domains. So, for each element, the browser (and its
³³⁸ supporting O/S) must first resolve the corresponding domain name, then establish a connection to
³³⁹ the given server, and finally download and then render the provided content. Thus for each element
³⁴⁰ that needs to be displayed, the ΔQ is the sequential composition of the ΔQs of the component steps
³⁴¹ described above; and the ΔQ of rendering the whole page is an *all-to-finish* combination of the ΔQs
³⁴² of all the elements. Note that this formulation *automatically* deals with the possibility that any of the
³⁴³ steps may fail, and provides the resultant failure probability for the whole process in addition to the
³⁴⁴ distribution of expected completion times.

³⁴⁵   This simple model can be further refined, as needed to meet real-world requirements. For example,
³⁴⁶ DNS resolution might provide alternative server addresses for load-balancing purposes, and each
³⁴⁷ of these servers might have different ΔQs when providing the same content to the user (perhaps
³⁴⁸ because they are located in different geographical locations, or are provisioned using systems with
³⁴⁹ different CPU or storage capabilities). We can represent this as a probabilistic choice between these
³⁵⁰ outcomes, weighted by the probability that a specific server is used. This weights the corresponding
³⁵¹ ΔQ. In addition, we might also consider the effect of load and contention for shared resources, for
³⁵² example network interface bandwidth or rendering capacity, or the impact of different DNS caching
³⁵³ architectures on performance. These aspects of system performance design are formalised in Section 5.

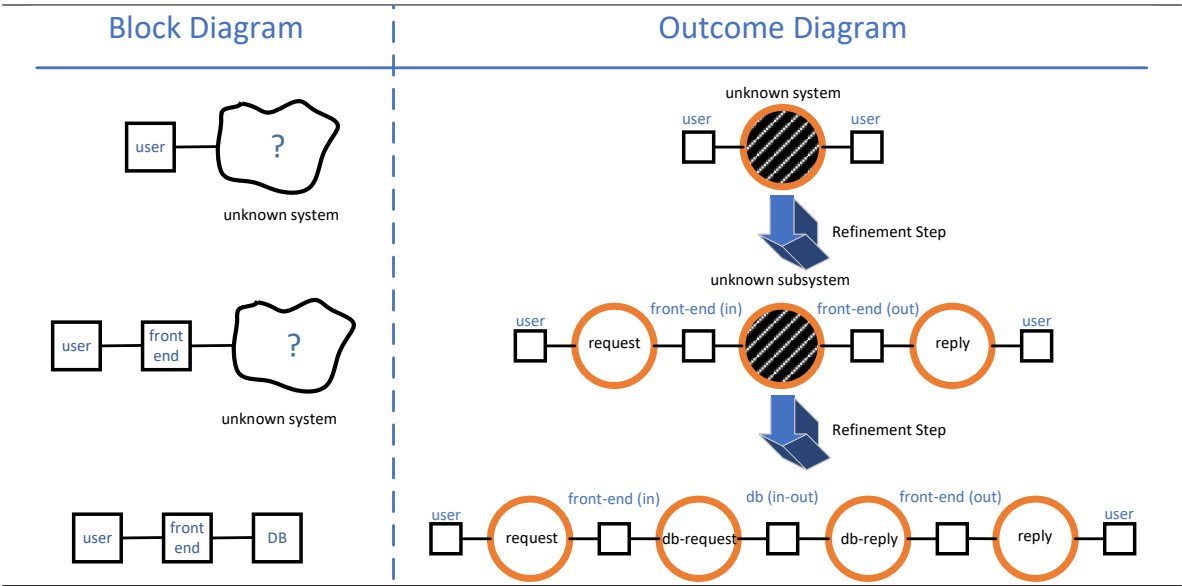



Figure 5. Block Diagram and Outcome Diagram for a Simple System Constructed using Stepwise Refinement.

3.5. Alternatives to Outcome Diagrams — Why a New Diagram?

The ΔQSD paradigm introduces the concept of outcome diagrams. It is perfectly reasonable to ask at this point: "Why another diagram? What is it that outcome diagrams capture that UML diagrams, for example, cannot?" Let us answer these questions by comparing outcome diagrams with UML. We first recall the two main properties of outcome diagrams in the ΔQSD paradigm:

- An outcome diagram specifies the *causal relations between outcomes*. An outcome is a specific system behaviour defined by its possible starting events and its possible terminating events. For example, sending a message to a server is an outcome defined by the beginning of the send operation and the end of the send operation. The action of sending a message and receiving a reply is observed as an outcome, defined by the beginning of the send operation and the end of receive operation. Outcomes can be decomposed into smaller outcomes, and outcomes can be causally related. For example, the send-receive outcome can be seen as a causal sequence of a send outcome and a receive outcome.
- An outcome diagram can be defined for a *partially specified system*. Such an outcome diagram can contain undefined outcomes, which are called black boxes. A black box does not correspond to any defined part of the system, but it still has timeliness and resource constraints. Refining an outcome diagram can consist in replacing one of its black boxes with a subgraph of outcomes.

A crucial property of an outcome diagram is that it is an *observational* concept. That is, it says something about **what** a user observes of a system from the **outside**, but it does not say anything about how the system is constructed internally.

3.5.1. UML Diagrams

UML is a rich language defined to model many different aspects of software, including its structure, behavior, and the processes it is part of. The UML 2 standard defines 14 kinds of diagrams, which are classified into structural diagrams and behavioural diagrams. We first note two general properties of outcome diagrams that UML diagrams do not share:

- *Observational property* All UML diagrams, structural and behavioural, define what happens inside the system being modeled, whereas outcome diagrams define observations from outside the system. The outcome diagram makes no assumptions about the system's components or internal states.
- *Wide coverage property* It is possible for both UML diagrams and outcome diagrams to give partial information about a system, so that they correspond to many possible systems. As long as the systems are consistent with the information in the diagram, they will have the same diagram. However, an outcome diagram corresponds to a much larger set of possible systems than a UML diagram. For an outcome diagram, a system corresponds if it has the same outcomes, independent of its internal structure or behaviour. For a UML diagram, a system corresponds if its internal structure or behaviour is consistent with the information in the diagram. This means that a UML diagram is already making decisions w.r.t. the possible system structures quite early in the design process. The outcome diagram does not make such decisions.

In the rest of this section, we compare outcome diagrams to two UML diagrams, namely the state machine diagram and the component diagram.

3.5.2. State Machine Diagram

A state machine diagram is a finite state automaton. It defines the internal states of a system and the transitions between them. The state diagram captures the causality between the actions taken when the system changes states, but this does not map directly to the outcomes observed by an external user. There is however a relationship between a state diagram and an outcome diagram. An outcome can map to a sequence of state transitions; whereas, by examining the actions of a state diagram, it is possible to deduce the outcomes to expect from taking those actions.

3.5.3. Block Diagram

A block diagram specifies a system as a set of elements with their interconnections. We illustrate the difference between block diagrams and outcome diagrams using a simple example system, a user querying a front end that is connected to a database (Figure 5). The figure shows the refinement process: a system with an initially unknown structure is refined stepwise into a system that has a completely known structure. For the outcome diagram, the system performance can be obtained directly by composing the ΔQs of the outcomes, using the rules described in Section 5. For the block diagram, it is harder to obtain system performance. This is because the block diagram does not define the expected outcomes of a system or their causality. The block diagram by itself does not have sufficient information to allow system performance to be calculated: we also need to know the expected outcome and the sequence of messages sent between blocks needed to achieve that outcome. As a final remark, the block diagram constrains the system structure to always have a front end and a database, whereas the outcome diagram is consistent with many alternative system structures.

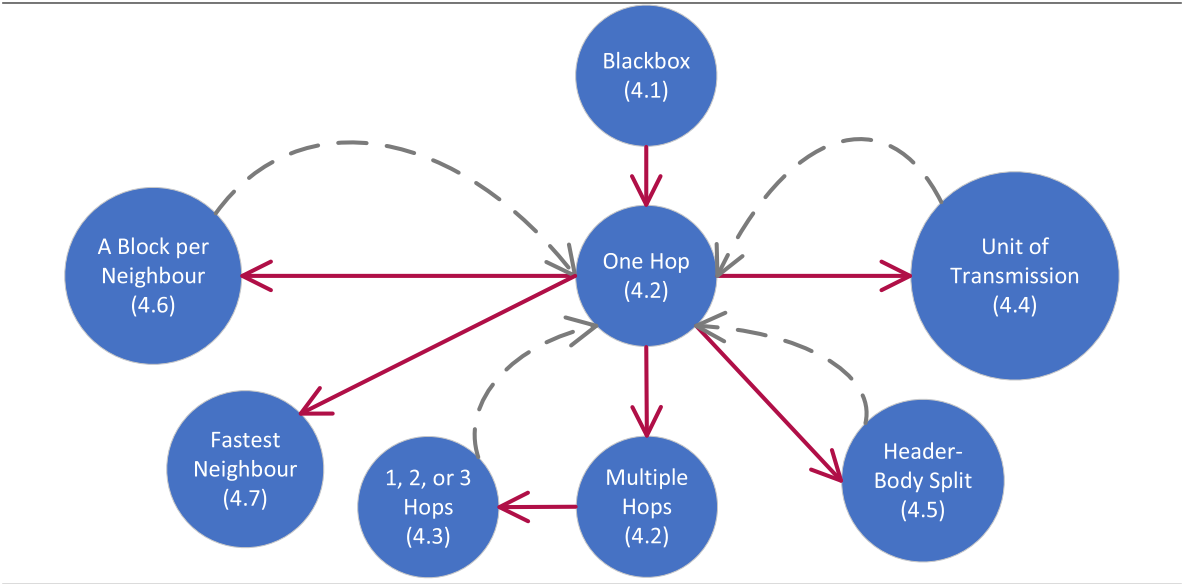



Figure 6. Design Exploration of a Blockchain Diffusion example, as described in Subsections 4.1–4.7. Solid lines are for refinement; dashed lines are for backtracking to coarser granularity.

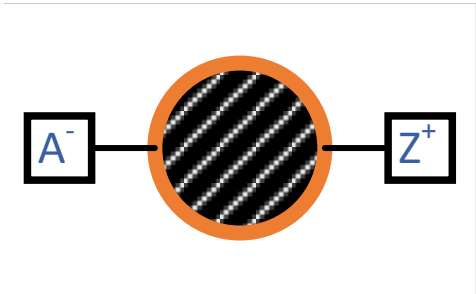
4. Design Exploration using Outcome Diagrams

This section simulates how a design engineer could explore the blockchain diffusion example that was described in Section 2, using outcome diagrams. Figure 6 depicts that design exploration in the form of a threaded decision tree in the search space. Each node in the tree is an outcome diagram. Every node is labelled with a description plus the section in this paper where it is discussed. There are two types of edges: solid edges represent refinement steps (Definition 7); whilst dashed edges represent backtracks to take alternative directions of refinement. The formalism used in this section is presented in Section 5.

4.1. Starting Off

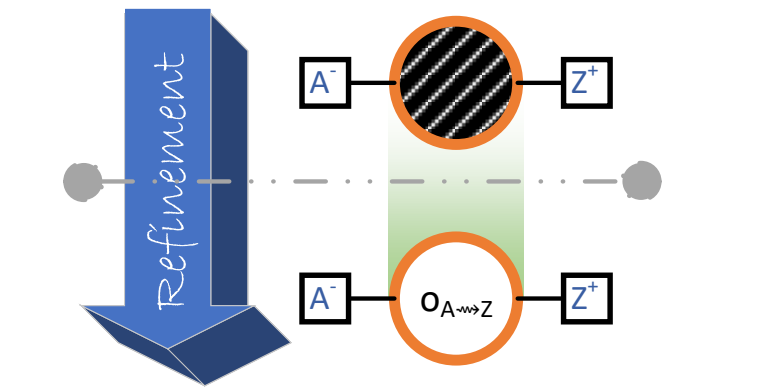
Initially, the design engineer knows almost nothing about the system. Perhaps, all they know is that there will be the following two observation locations:

- A^-: Block is ready to be transmitted by A.
- Z^+: Block is received and verified by Z.

The corresponding outcome diagram is

in which the only outcome is a black box. As will be detailed in Section 5, the outcome expression to describe that outcome diagram is a ♭ (♭ for black boxes).

431 *4.2. Early Analysis*

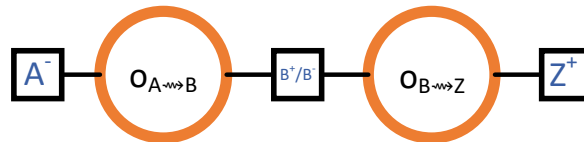
432   Given that the design engineer is not content with the current level of granularity, they wish to
433 further detail the diagram by giving the black box a name, such as $o_{A \rightsquigarrow Z}$. In ΔQSD, we call adding
434 that further detail a *refinement*. That refinement step is depicted below:

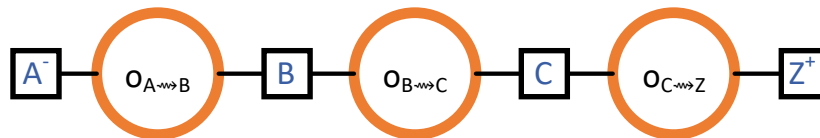
435

Here, the outcome diagram that is above the dashed line is refined into the one below. As will be
discussed in Section 5, the (rewrite) rule that authorises this refinement is

$$\mathcal{C}[\flat] \rightarrow \mathcal{C}[o].$$

436 We call this rule (UNBX) for unboxing (a black box). The rule states that, in a context $\mathcal{C}$, a black box
437 can be rewritten to any other outcome expression (but not to a black box). In this case, we choose the
438 black box to be rewritten to an *outcome variable* called $o_{A \rightsquigarrow Z}$. This indicates the outcome of hopping
439 directly from A to Z.

440   Before producing more of our block diffusion algorithm's outcome diagram, we would like to take
441 the time to apply some analysis. Refinements aside, suppose for a moment that there are two hops to
442 make from A to Z: first from A to an intermediate node B; and, then, from B to Z. The corresponding
443 outcome diagram for the two-hop journey from A to Z would then be:

444

445 Here, $o_{A \rightsquigarrow B}$ and $o_{B \rightsquigarrow Z}$ are the outcomes of hopping from A to B and from B to Z, respectively. Note
446 also that the observation location between the above two outcomes is labelled B^+/B^-. That is because
447 the observation B^+ and B^- take place at the same location. For that reason, we will simply write B to
448 refer to that observation location. The same convention is used for similar intermediate locations. It is
449 then easy to obtain the outcome diagram for three hops:

450

    While outcome diagrams are visually more attractive; outcome expressions are algebraically more
attractive. For example, the corresponding expression for two hops is $o_{A \rightsquigarrow B} \bullet\!\!\rightarrow\!\!\bullet o_{B \rightsquigarrow Z}$, where "$\bullet\!\!\rightarrow\!\!\bullet$" is
the symbol we use for *sequential composition*: The sequential composition of $o_{A \rightsquigarrow B}$ and $o_{B \rightsquigarrow Z}$ is needed
because the latter causally depends on the former. Likewise, the outcome expression for three hops is
$o_{A \rightsquigarrow B} \bullet\!\!\rightarrow\!\!\bullet o_{B \rightsquigarrow C} \bullet\!\!\rightarrow\!\!\bullet o_{C \rightsquigarrow Z}$. Generalising that to n hops then is easy: $o_{A \rightsquigarrow B_1} \bullet\!\!\rightarrow\!\!\bullet o_{B_1 \rightsquigarrow B_2} \bullet\!\!\rightarrow\!\!\bullet \ldots \bullet\!\!\rightarrow\!\!\bullet$
$o_{B_n \rightsquigarrow Z}$, which we abbreviate as $o_{A \rightsquigarrow B_1} \bullet\!\!\rightarrow\!\!\bullet \left({}^{\bullet\!\!\rightarrow\!\!\bullet\, n-1}_{\bullet\!\!\rightarrow\!\!\bullet\, 1} o_{B_i \rightsquigarrow B_{i+1}} \right) \bullet\!\!\rightarrow\!\!\bullet o_{B_n \rightsquigarrow Z}$. Parameterisation by n hops

Distance	time (s)	64kB		256kB		512kB		1024kB		2048kB	
		time(s)	RTTs	time(s)	RTTs	time(s)	RTTs	time(s)	RTTs	time(s)	RTTs
Short	0.012	0.024	*1.95*	0.047	*3.81*	0.066	*5.41*	0.078	*6.36*	0.085	*6.98*
Medium	0.069	0.143	*2.07*	0.271	*3.94*	0.332	*4.82*	0.404	*5.87*	0.469	*6.81*
Long	0.268	0.531	*1.98*	1.067	*3.98*	1.598	*5.96*	1.598	*5.96*	1.867	*6.96*

Table 1. Representative times in seconds and round-trip-times (RTTs) for one-way TCP transmission of varying block sizes for short, medium and long distances between blockchain nodes.

is useful because it helps the design engineer determine the right n for their blockchain. For example, a relevant question is: *What is the optimal n for block diffusion to be timely and for its load to be bearable?* The formalisation in Section 5 instructs the design engineer as to how to achieve that and other goals. Before detailing the how, we take our moment to analyse a smaller example. Consider the two-hop scenario. Provided that the design engineer has ΔQs for both $o_{A \rightsquigarrow B}$ and $o_{B \rightsquigarrow Z}$, they can use Definition 4 to work out the ΔQ of $o_{A \rightsquigarrow B} \bullet{\rightarrow}\bullet o_{B \rightsquigarrow Z}$:

$$\Delta Q(o_{A \rightsquigarrow B} \bullet{\rightarrow}\bullet o_{B \rightsquigarrow Z}) = \Delta Q(o_{A \rightsquigarrow B}) * \Delta Q(o_{B \rightsquigarrow Z}).$$

451   In a similar vein, the design engineer can work out the n-hop scenario's ΔQ for $n > 1$:

$$\Delta Q(o_{A \rightsquigarrow B_1} \bullet{\rightarrow}\bullet (\overset{\bullet\rightarrow\bullet^{n-1}}{\bullet\rightarrow\bullet_{i=1}} o_{B_i \rightsquigarrow B_{i+1}}) \bullet{\rightarrow}\bullet o_{B_n \rightsquigarrow Z}) =$$
$$\Delta Q(o_{A \rightsquigarrow B_1}) * \overset{n-1}{\underset{i=1}{*}} \Delta Q(o_{B_i \rightsquigarrow B_{i+1}}) * \Delta Q(o_{B_n \rightsquigarrow Z}). \quad (2)$$

452

453   Then, using the formulation given in Definition 5, the design engineer can determine the constraints
454   on n that are needed in order for block diffusion to meet the overall timeliness requirements.
455   In practice, the time that is needed to transfer a block of data one hop depends on four main
456   factors:

457   1. The size of the block;
458   2. The speed of the network interface;
459   3. The geographical distance of the hop (as measured by the time to deliver a single packet);
460   4. Congestion along the network path.

461   When we consider blockchain nodes that are located in data centres (which most block producers tend
462   to be), the interface speed will typically be 1Gb/s or more. This is not a significant limiting factor for
463   the systems of interest (see Section 5.4 for an analysis that explains this). In the setting that we are
464   considering, congestion is generally minimal, and so this can also be ignored in the first instance. This
465   leaves: i) block size, which we will take as a design parameter to be investigated later; and ii) distance,
466   which we will consider now. For simplicity, we will consider three cases of geographical distance:

467   **Short** : the two nodes are located in the same data centre;
468   **Medium** : the two nodes are located in the same continent;
469   **Long** : the two nodes are located in different continents.

470   For pragmatic reasons, Cardano relies on the standard TCP protocol for data transfers. TCP transforms
471   loss into additional delay, so the residual loss is negligible. At this point, we could descend into
472   a detailed refinement of the TCP protocol, but equally we could simply take measurements; the
473   compositionality of ΔQSD means that it makes no difference where the underlying values come from.
474   Table 1 shows measurements of the transit time of packets and the corresponding transfer time of
475   blocks of various sizes, using hosts running on AWS data centre servers in Oregon, Virginia, London,
476   Ireland and Sydney. Since we know that congestion is minimal in this setting, the spread of values will
477   be negligible, and so in this case the CDFs for the ΔQs will be step functions. The transfer time for

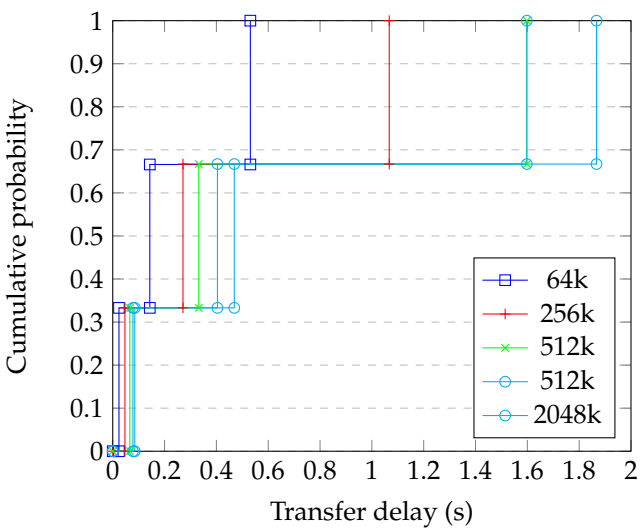
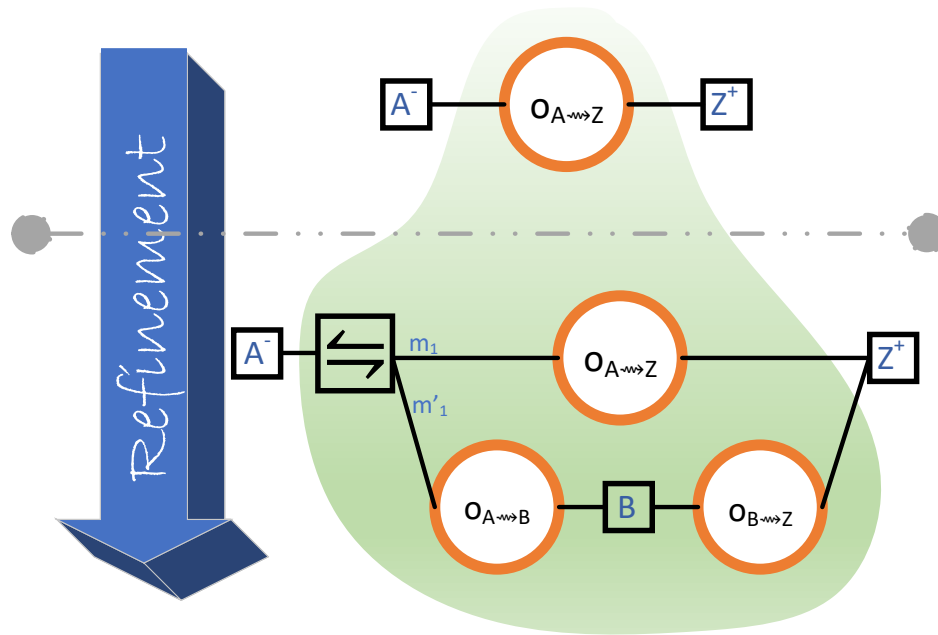



Figure 7. One-hop delay distributions per block size

each block size is given both in seconds and in multiples of the basic round-trip time (RTT) between the hosts in question. Since the TCP protocol relies on the arrival of acknowledgements to permit the transmission of more data, it is unsurprising to see a broadly linear relationship, which could be confirmed by a more detailed refinement of the details of the protocol.

Given the randomness in the network structure and the selection of block-producing nodes, there remains some uncertainty on the length of an individual hop. At this point, we will assume that short, medium and long hops are equally likely, which we can think of as an equally-weighted probabilistic choice. In numerical terms, this becomes a weighted sum of the corresponding ΔQs, as given in Table 1. This gives the distribution of transfer times per block size shown in figure 7.

4.3. Refinement and Probabilistic Choice

Recall that A and Z are names for randomly chosen nodes, so the number of hops between A and Z is unkown. ΔQSD tackles that uncertainty by offering an outcome diagram which involves probabilistic choice between the different number of hops that might be needed. Strictly speaking, a probabilistic choice is a binary operation. Hence, when there are more than two choices the outcome diagram will cascade probabilistic choices. In the general formulation, there are at most n hops. In order to produce that, the design engineer exercises a step-by-step refinement of the single-hop outcome diagram. The first refinement introduces the choice between one or two hops:

The equivalent outcome expression is $o_{A\rightsquigarrow Z} \underset{m'_1}{\overset{m_1}{\rightleftharpoons}} (o_{A\rightsquigarrow B} \bullet\!\!\rightarrow\!\!\bullet o_{B\rightsquigarrow Z})$, a probabilistic choice between one or two hops with respective weights m_1 and m'_1. The corresponding (rewrite) rule is:

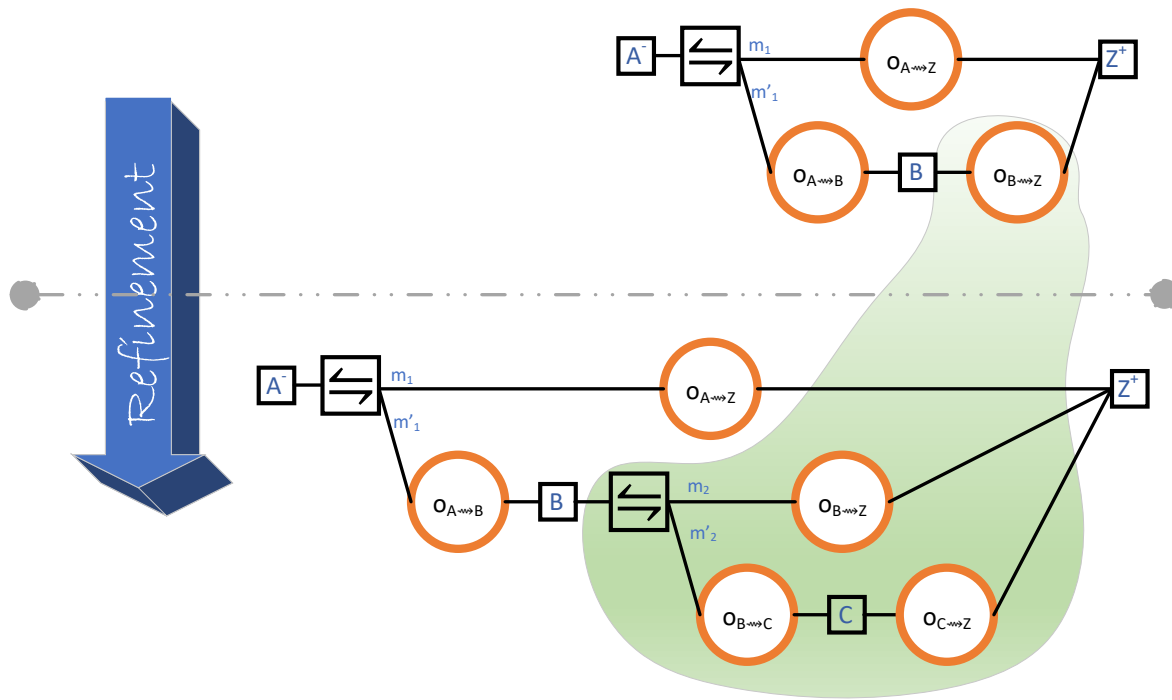
$$\mathcal{C}[o] \rightarrow \mathcal{C}[o' \underset{m''}{\overset{m'}{\rightleftharpoons}} o'']$$

which we call (PROB) (for probabilistic choice). Here is how we applied (PROB) to arrive from the single hop to the probabilistic choice between one hop and two hops:

$$o_{A\rightsquigarrow Z} \rightarrow o_{A\rightsquigarrow Z} \underset{m'_1}{\overset{m_1}{\rightleftharpoons}} (o_{A\rightsquigarrow B} \bullet\!\!\rightarrow\!\!\bullet o_{B\rightsquigarrow Z})$$

That is, $\mathcal{C}$ in the above refinement is an empty *context*.

Next, the design engineer further refines the two-hop part to the probabilistic choice between two or three hops.

499

We use (PROB) again. However, instead of an empty context, here the context is $o_{A \leadsto Z} \overset{m_1}{\underset{m'_1}{\rightleftharpoons}} (o_{A \leadsto B} \bullet \to \bullet$ $[.])$:

$$o_{A \leadsto Z} \overset{m_1}{\underset{m'_1}{\rightleftharpoons}} (o_{A \leadsto B} \bullet \to \bullet \ o_{B \leadsto Z}) \to o_{A \leadsto Z} \overset{m_1}{\underset{m'_1}{\rightleftharpoons}} (o_{A \leadsto B} \bullet \to \bullet \ (o_{B \leadsto Z} \overset{m_2}{\underset{m'_2}{\rightleftharpoons}} (o_{B \leadsto C} \bullet \to \bullet o_{C \leadsto Z}))).$$

500 The design engineer can continue refinement until a predetermined number of hops is reached.
501 Alternatively, they can keep the number of hops as a parameter and analyse the corresponding
502 parameterised outcome expression for timeliness, behaviour under load, etc.

503   Figure 8 shows the result of applying equation 2 to the sequence of outcome expressions
504 corresponding to one, two, ... five sequential hops using the transfer delay distribution shown
505 in Figure 7, for a 64kB block size. It can be seen that there is a 95% probability of the block arriving
506 within 2s. In contrast, Figure 9 shows the corresponding sequence of delay distributions for a 1024kB
507 block size, where the 95th percentile of transfer time is more than 5s.

508   If we know the distribution of expected path lengths, we can combine the ΔQs for different hop
509 counts using (PROB). Table 2 shows the distribution of paths lengths in simulated random graphs
510 having 2500 nodes and a variety of node degrees [18]. Using the path length distribution for nodes of
511 degree 10, for example, then gives the transfers delay distribution shown in Figure 10.

Length	Node degree			
	5	10	15	20
1	0.20	0.40	0.60	0.80
2	1.00	3.91	8.58	14.72
3	4.83	31.06	65.86	80.08
4	20.18	61.85	24.95	4.40
5	47.14	2.78	0.00	
6	24.77	0.00		
7	1.83			
8	0.05			

Table 2. Percentage of paths having a given length in a random graph of 2500 nodes of varying degree

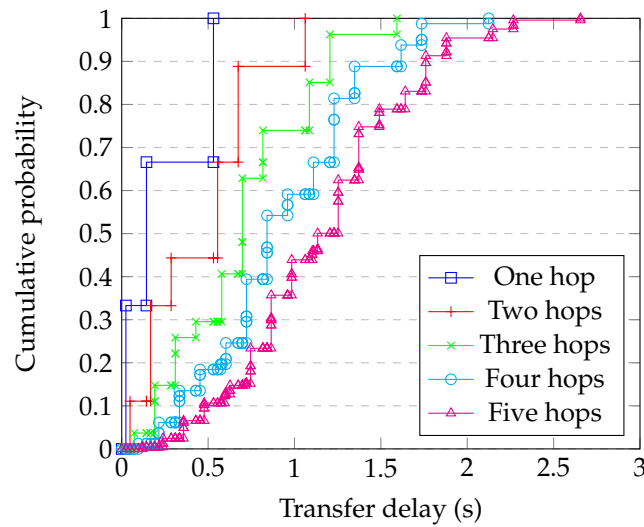



Figure 8. Multi-hop delay distributions for 64k block size

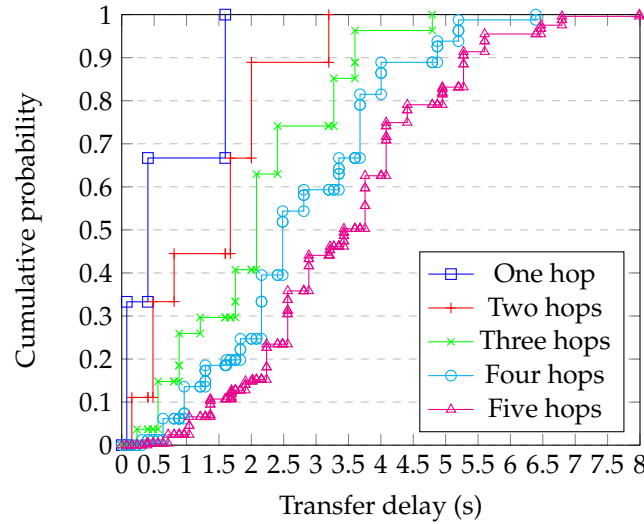



Figure 9. Multi-hop delay distributions for 1024k block size

4.3.1. Alternative Refinements

Suppose that, instead of investigating the number of hops, the design engineer is now interested in studying the steps within a single hop. There are various ways to do this. In Sections 4.4–4.7, we will consider four different ways that can be used when A and Z are neighbours. each of which refines $o_{A \rightsquigarrow Z}$. These refinements are all instances of the (ELAB) (rewrite) rule (for elaboration):

$$\mathcal{C}[o_v] \rightarrow \mathcal{C}[o].$$

The following sections are also important for another reason. So far, we have traversed the threaded tree of refinement in a depth-first way; the upcoming subsections traverse that tree in a breadth-first way. ΔQSD allows the design engineer to choose between depth-first and breadth-first refinement at any point in their design exploration.

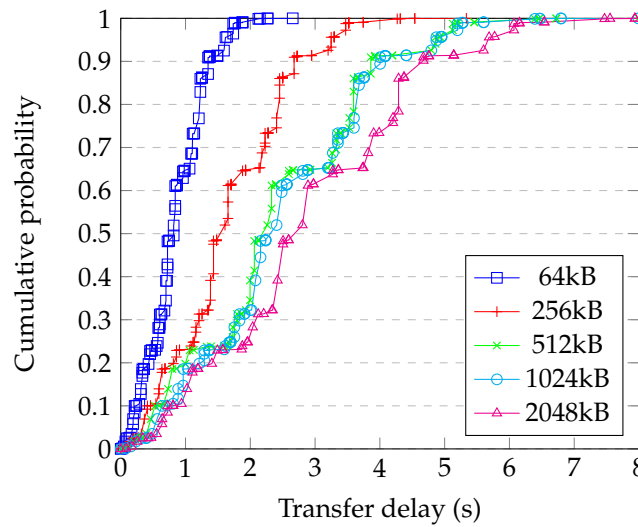



Figure 10. Multi-hop delay distributions for varying block size in a graph of 2500 nodes with node degree 10

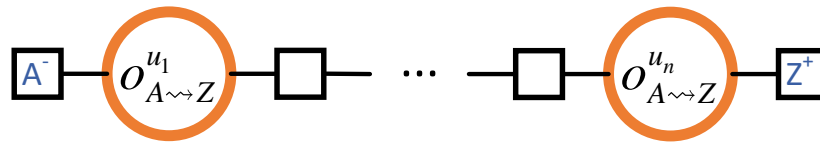



Figure 11. Breaking down the Transmission of a Message into n Smaller Units.

4.4. Breaking Down Transmissions into Smaller Units

Network transmissions are typically broken down into the transmission of smaller *unit*s. Depending on the layering of the network protocols, that might, for example mean dividing a high-level message into several smaller packets. In a similar vein, the design engineer might decide to study block diffusion in terms of smaller units of transmission. For example, they might want to study the division of $o_{A \rightsquigarrow Z}$ into n smaller unit operations $o_{A \rightsquigarrow Z}^{u_1}, \ldots, o_{A \rightsquigarrow Z}^{u_n}$. The resulting outcome diagram is shown in Figure 11. The corresponding outcome expression would then be $o_{A \rightsquigarrow Z} \rightarrow o_{A \rightsquigarrow Z}^{u_1} \bullet \rightarrow \bullet \ldots \bullet \rightarrow \bullet o_{A \rightsquigarrow Z}^{u_n}$, which we abbreviate as $o_{A \rightsquigarrow Z} \rightarrow \underset{1}{\overset{n}{\bullet \rightarrow \bullet}} o_{A \rightsquigarrow Z}^{u_i}$. This refinement can happen at different levels of granularity and is fairly repetitive. However, this is the level at which details of the transmission protocol such as TCP could be introduced if required.

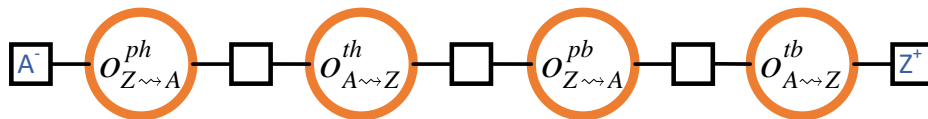



Figure 12. Splitting a Block Transmission into its Constituent Parts: Header (*ph/th*) and Body (*pb/tb*).

4.5. Header-Body Split

In Cardano Shelley, an individual block transmission involves a dialogue between a sender node, A, and a recipient node, Z. We represent the overall transmission as $o_{A \rightsquigarrow Z}$. This can be refined into the following sequence:

1. *Permission for Header Transmission ($o_{Z \rightsquigarrow A}^{ph}$):* Node Z grants the permission to node A to send it a header.
2. *Transmission of the Header ($o_{A \rightsquigarrow Z}^{th}$):* Node A sends a header to node Z.
3. *Permission to for Body Transmission ($o_{Z \rightsquigarrow A}^{pb}$):* Node Z analyses the header that was previously sent to it by A. Once the suitability of the block is determined via the header, node Z grants permission to A to send it the respective body of the previously sent header.
4. *Transmission of the Body ($o_{A \rightsquigarrow Z}^{tb}$):* Finally, A sends the block body to Z.

The motivation for the header/body split and the consequential dialogue is optimisation of transmission costs. Headers are designed to be affordably cheap to transmit. In addition, they carry enough information about the body to enable the recipient to verify its suitability. The body is only sent once the recipient has done this. This prevents the unnecessary transmission of block bodies when they are not required. Since bodies are typically many orders of magnitude larger than headers, considerable network bandwidth can be saved in this way. Moreover, the upstream node is not permitted to send another header until given permission to do so by the downstream node, in order to prevent a denial-of-service attack in which a node is bombarded with fake headers, so this approach also reduces latency when bodies are rejected. In practice, the first permission is sent when the connection between peers is established, and the permission renewed immediately after the header is received, so that the upstream peer does not have to wait unnecessarily. The design engineer can therefore refine $o_{A \rightsquigarrow Z}$ into the finer grained outcomes shown in Figure 12. The corresponding outcome expression is $o_{Z \rightsquigarrow A}^{ph} \bullet\!\!+\!\!\bullet o_{A \rightsquigarrow Z}^{th} \bullet\!\!+\!\!\bullet o_{Z \rightsquigarrow A}^{pb} \bullet\!\!+\!\!\bullet o_{A \rightsquigarrow Z}^{tb}$.

Note that the protocol described here is between directly connected neighbours - these requests are not forwarded to other nodes. This is thus a refinement of the one-hop block transfer process. The significance of this refinement is that it shows that an individual outcome that, at a given level of granularity, is unidirectional (i.e., only from one entity in the system to another) might, at a lower level of granularity, very well be a multi-directional conversation.

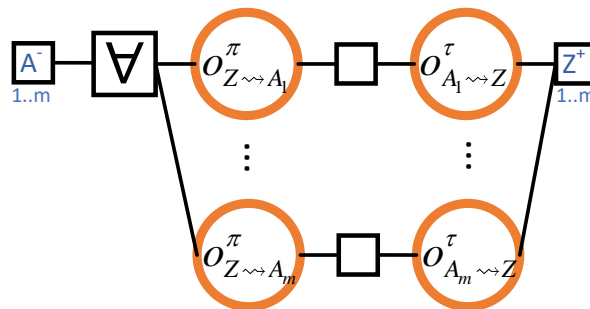



Figure 13. Obtaining one Block from each Neighbour when Rejoining the Blockchain.

4.6. Obtaining one Block from each Neighbour when Rejoining the Blockchain

Consider the situation where a node Z rejoins the blockchain after being disconnected for some period of time. Z will be out-of-date w.r.t. the recently generated blocks and will need to update itself. Let us consider the lucky situation where Z can acquire all the blocks that it is missing from its

560 neighbours; that is, it can acquire the blocks with only one hop but from different neighbours. For
561 demonstration purposes, we now make a number of simplifying assumptions:

562 • Upon its return to the blockchain, Z is m blocks behind, where m is less than or equal to the
563   number of Z's neighbours.
564 • Each neighbour A_i of Z transmits precisely one block to Z.
565 • The header-body split refinement of Section 4.5 is not considered. Therefore, there are only two
566   steps (instead of the actual four):

567   1. $o_{Z \leadsto A_i}^{\pi}$ for when Z grants $p$ermission to A_i. And,
568   2. $o_{Z \leadsto A_i}^{\tau}$ for when A_i $t$ransmits the (entire) block to Z.

With those simplifications in place, the outcome diagram will be as shown in Figure 13. This shows
that Z will be up-to-date when **all** its m (selected) neighbours are granted permission and have finished
sending their blocks to Z. Note that the outcome diagram has, in fact, m starting observation locations
and m terminating observation locations. This is the reason for the $1..m$ notation immediately below
each of those observation locations. The corresponding outcome expression is

$$\forall((o_{Z \leadsto A_1}^{\pi} \bullet \!\rightarrow\! \bullet\, o_{A_1 \leadsto Z}^{\tau}) \parallel \cdots \parallel (o_{Z \leadsto A_m}^{\pi} \bullet \!\rightarrow\! \bullet\, o_{A_m \leadsto Z}^{\tau}))$$

569 which we abbreviate as $\parallel^{\forall}(o_{Z \leadsto A_i}^{\pi} \bullet \!\rightarrow\! \bullet\, o_{A_i \leadsto Z}^{\tau})_1^m$.
570   One reason why this refinement is particularly interesting is that it allows an easy demonstration
571 of our load analysis from Section 5.4. Fix a resource ρ such as network capacity. Pick a time t between
572 the first observation made at an A_i^- and the last observation made at a Z_i^+. According to Definition 10,
573 the static amount of work S at time t that is required for performing $\parallel^{\forall}(o_{Z \leadsto A_i}^{\pi} \bullet \!\rightarrow\! \bullet\, o_{A_i \leadsto Z}^{\tau})_1^m$ is the
574 sum of the static amounts of work S that is required at t for performing each $o_{Z \leadsto A_i}^{\pi} \bullet \!\rightarrow\! \bullet\, o_{A_i \leadsto Z}^{\tau}$ (where
575 $1 \le i \le m$):

$$S[\parallel^{\forall}(o_{Z \leadsto A_i}^{\pi} \bullet \!\rightarrow\! \bullet\, o_{A_i \leadsto Z}^{\tau})_{i=1}^m](t) = \sum_{i=1}^{m} S[o_{Z \leadsto A_i}^{\pi} \bullet \!\rightarrow\! \bullet\, o_{A_i \leadsto Z}^{\tau}](t). \tag{3}$$

577 Equation (3) describes an approach to aggregating offered load on a resource. Consider an ephemeral
578 resource — such as a communications network interface — a design interest might be to understand
579 the intensity of use of this interface. For example, for a design requirement to be (at this level of
580 detail) feasible, the average use of the interface has to be less than its capacity. This is the basic
581 precondition for the demand on the resource to possess a feasible schedule. The RHS of equation (3)
582 captures this process as a piece-wise summation of the load intensities. Building on the time to transfer
583 blocks (Table 1), and noting (from Section 2.1) that the body of a block is forwarded in response to
584 a request (which takes one round trip time); the total block volume is delivered in the total time
585 minus the round trip time. For the 'Near' peers shipping a $64kB$ block, this means an intensity of
586 $42.7Mb/s(8*64e3/(0.024-0.012))$ *before* incorporating any other network related overheads (such as
587 layered headers). Table 3 captures that load intensity approximation.

	Block size (kB)				
Distance	64	256	512	1,024	2,048
Short	42.7	58.5	75.9	151.7	224.4
Medium	6.9	10.1	15.6	31.1	41.0
Long	1.9	2.6	3.1	6.2	10.2

Table 3. Average load intensities (in Mbit/s) implied by time to load from Table 1

588 This provides an insight into the likely capacity constraints for differing degrees of connectivity and, by
589 inference, an insight into the system level design trades. From Tables 1 and 3 it can be seen that smaller

geographic distribution can lead to lower forwarding times assuming that (for a fixed communications capacity) the number of associating peers is suitably reduced. Assessments such as this give a measure of the likely "slack" in the design; those portions of the design that have less "slack" represent design elements that might need more detailed refinement and/or other strategies to ensure their feasibility. Note that a dedicated support tool for ΔQSD would easily be able to manipulate these complex outcome diagrams, giving a formally correct analysis, with very little mental burden for the design engineer.

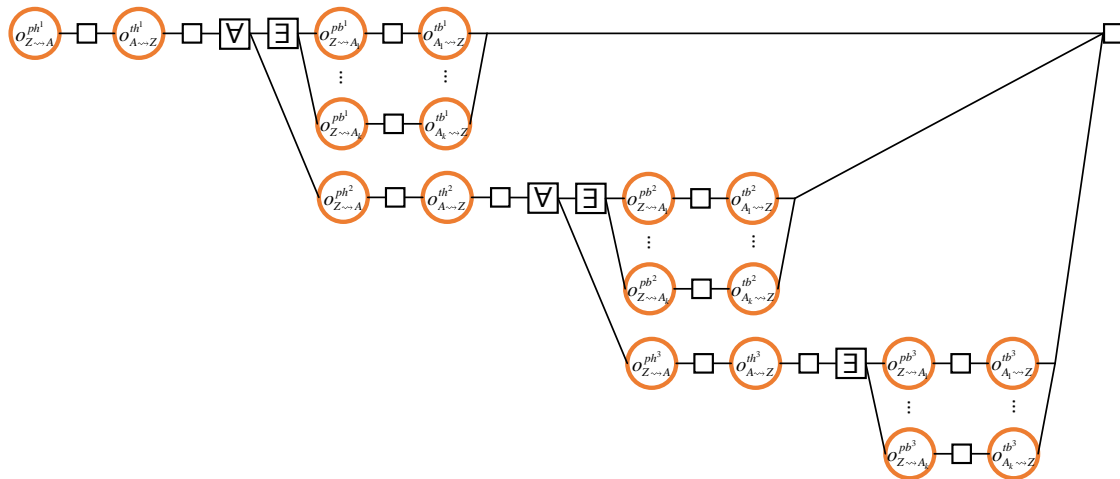



Figure 14. Obtaining a Block from the Fastest Neighbour.

4.7. Obtaining a Block from the Fastest Neighbour

Section 4.5 discussed splitting the header and body for optimisation reasons. One assumption in that design is that the header and the body will be taken from the same neighbour. It turns out that this assumption will not necessarily lead to the fastest solution. In fact, when Z determines that it is interested in a block that it has received the header of, it may obtain it from *any* of its neighbours that have signalled that they have it. In particular, Cardano nodes keep a record of the ΔQs of their neighbours' block delivery. This allows them to obtain bodies from their fastest neighbour(s). In other words, once a node determines the desirability of a block (via its header), it is free to choose to take the body from any of its neighbours that have provided the corresponding header. As long as only timeliness is a concern – and not when resource consumption is also of interest – a *race* can occur between **all** neighbours, with the fastest neighbour winning the race. The diagrams in this section assume such a race.

Now, like Section 4.6, consider the situation where Z reconnects to the blockchain after being disconnected for some time. Our design in Section 4.6 assumes that there is no causality between the m blocks that Z needs to obtain. In reality, that is not correct: there is a causal order between those blocks, and that order can be rather tricky to define; it might take a couple of reads before the matter is fully digested. There are two separate total orders between blocks:

CO1. For each block, the header must be transmitted before the body (so that the recipient node can determine the suitability of the block before the body transmission);

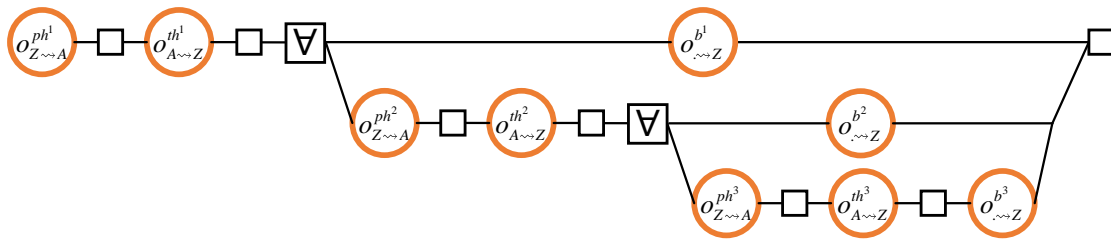
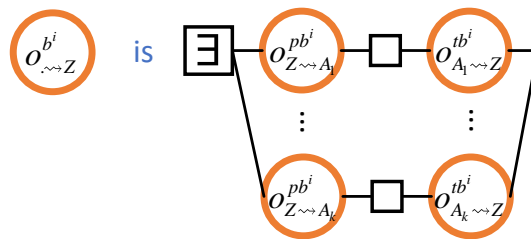
CO2. Headers of the older blocks need to be transmitted before those of the younger blocks (*note, however, that there is no causal relationship between the body transmissions*).

This section considers the situation when the design engineer investigates the above race as well as CO1 and CO2. Suppose that once Z reconnects to the blockchain, it is exactly $m = 3$ blocks behind the

current block. Suppose also that Z has k neighbours. The corresponding outcome diagram is shown in Figure 14. The fork that is causally dependent on $o_{A \rightsquigarrow Z}^{th^3}$ is done when **any** of its prongs is done, that is, as soon as any neighbour of Z has finished transmitting the third block to Z. The other "$\exists$" forks are similar.

The corresponding outcome expression is:

$$o_{Z \rightsquigarrow A}^{ph^1} \bullet\!\!\rightarrow\!\!\bullet o_{A \rightsquigarrow Z}^{th^1} \bullet\!\!\rightarrow\!\!\bullet \forall (\exists [(o_{Z \rightsquigarrow A_1}^{pb^1} \bullet\!\!\rightarrow\!\!\bullet o_{A_1 \rightsquigarrow Z}^{tb^1}) \parallel \cdots \parallel (o_{Z \rightsquigarrow A_k}^{pb^1} \bullet\!\!\rightarrow\!\!\bullet o_{A_k \rightsquigarrow Z}^{tb^1})] \parallel$$
$$(o_{Z \rightsquigarrow A}^{ph^2} \bullet\!\!\rightarrow\!\!\bullet o_{A \rightsquigarrow Z}^{th^2} \bullet\!\!\rightarrow\!\!\bullet \forall (\exists [(o_{Z \rightsquigarrow A_1}^{pb^2} \bullet\!\!\rightarrow\!\!\bullet o_{A_1 \rightsquigarrow Z}^{tb^2}) \parallel \cdots \parallel (o_{Z \rightsquigarrow A_k}^{pb^2} \bullet\!\!\rightarrow\!\!\bullet o_{A_k \rightsquigarrow Z}^{tb^2})] \parallel$$
$$(o_{Z \rightsquigarrow A}^{ph^3} \bullet\!\!\rightarrow\!\!\bullet o_{A \rightsquigarrow Z}^{th^3} \bullet\!\!\rightarrow\!\!\bullet \exists [(o_{Z \rightsquigarrow A_1}^{pb^3} \bullet\!\!\rightarrow\!\!\bullet o_{A_1 \rightsquigarrow Z}^{tb^3}) \parallel \cdots \parallel (o_{Z \rightsquigarrow A_k}^{pb^3} \bullet\!\!\rightarrow\!\!\bullet o_{A_k \rightsquigarrow Z}^{tb^3})])]))).$$

We would like to invite the reader to take their time to pair the above diagram with our explanations above. We understand that the diagram and to a greater degree the expression can look impenetrable. Compositionality of our formalism (inherited from that of ∆QSD) comes to rescue! Indeed, we can observe that the race pattern is rather repetitive. Thus, we can wrap the entire race into three new outcomes $o_{. \rightsquigarrow Z}^{b^1}$, $o_{. \rightsquigarrow Z}^{b^2}$, and $o_{. \rightsquigarrow Z}^{b^3}$. The intention is for $o_{. \rightsquigarrow Z}^{b^1}$, for example, to be the outcome of obtaining the first body transmitted to Z by **any one of its k neighbours** (that is, we are using "." in the subscript of $o_{. \rightsquigarrow Z}^{b^1}$ as a wildcard). This makes the outcome diagram considerably simpler:

where

These new diagrams make it easy to spot the lack of causal relationship between the $o_{. \rightsquigarrow Z}^{b^i}$s. Hence, there is no causal order between the body transmission despite the existence of CO1 and CO2. The corresponding outcome expression also becomes considerably simpler:

$$o_{Z \rightsquigarrow A}^{ph^1} \bullet\!\!\rightarrow\!\!\bullet o_{A \rightsquigarrow Z}^{th^1} \bullet\!\!\rightarrow\!\!\bullet \forall (o_{. \rightsquigarrow Z}^{b^1} \parallel (o_{Z \rightsquigarrow A}^{ph^2} \bullet\!\!\rightarrow\!\!\bullet o_{A \rightsquigarrow Z}^{th^2} \bullet\!\!\rightarrow\!\!\bullet \forall (o_{. \rightsquigarrow Z}^{b^2} \parallel (o_{Z \rightsquigarrow A}^{ph^3} \bullet\!\!\rightarrow\!\!\bullet o_{A \rightsquigarrow Z}^{th^3} \bullet\!\!\rightarrow\!\!\bullet o_{. \rightsquigarrow Z}^{b^3}))))$$

where

$$o_{. \rightsquigarrow Z}^{b^i} = \exists [(o_{Z \rightsquigarrow A_1}^{pb^i} \bullet\!\!\rightarrow\!\!\bullet o_{A_1 \rightsquigarrow Z}^{tb^i}) \parallel \cdots \parallel (o_{Z \rightsquigarrow A_k}^{pb^i} \bullet\!\!\rightarrow\!\!\bullet o_{A_k \rightsquigarrow Z}^{tb^i})]$$

which we abbreviate as

$$\parallel^{\exists} (o_{Z \rightsquigarrow A_j}^{pb^i} \bullet\!\!\rightarrow\!\!\bullet o_{A_j \rightsquigarrow Z}^{tb^i})_{j=1}^{m}.$$

The latter outcome diagrams and outcome expressions are now relatively easy to follow.

4.8. Summary

The refinements and analysis in this section have captured part of the design journey of Cardano Shelley. In Section 4.1, we defined a 'top level' outcome of interest, that of diffusing a block from an arbitrary source node to an arbitrary destination in a bounded time and with bounded resource consumption. In Section 4.2 we refined this to examine the implications of forwarding the block through a sequence of intermediate nodes, and in Section 4.3 we factored in the expected distribution of path lengths. This allows an exploration of the trade-offs between graph size, node degree, block size and diffusion time.

In Section 4.4 we showed how ΔQSD can be used to explore orthogonal aspects of the design, in this case how blocks of data are in fact transmitted as a sequence of packets. This could be extended into a full analysis of some transmission protocol such as TCP or QUIC. In Section 4.5 we analysed the effects of splitting blocks into a header and a body in order to reduce resource consumption, and in Section 4.6 the potential for speeding up block downloading by using multiple peers in parallel. An analysis of the network resource consumption in this case gave a flavour of how the ΔQSD paradigm encompasses resource as well as timeliness constraints.

In Section 4.7 we discussed how ΔQ is used in Cardano Shelley in operation as well as in design, to optimise the choice of peer from which to obtain a block.

All of this, together with further optimisations such as controlling the formation of the node graph to achieve a balance between fast block diffusion and resilience to partitioning, has produced an industry-leading blockchain implementation that functions reliably delivering a block up to 72kB every 20s on average. A snapshot of the 95th centile of block diffusion times over nearly 48 hours is given in Figure 15.

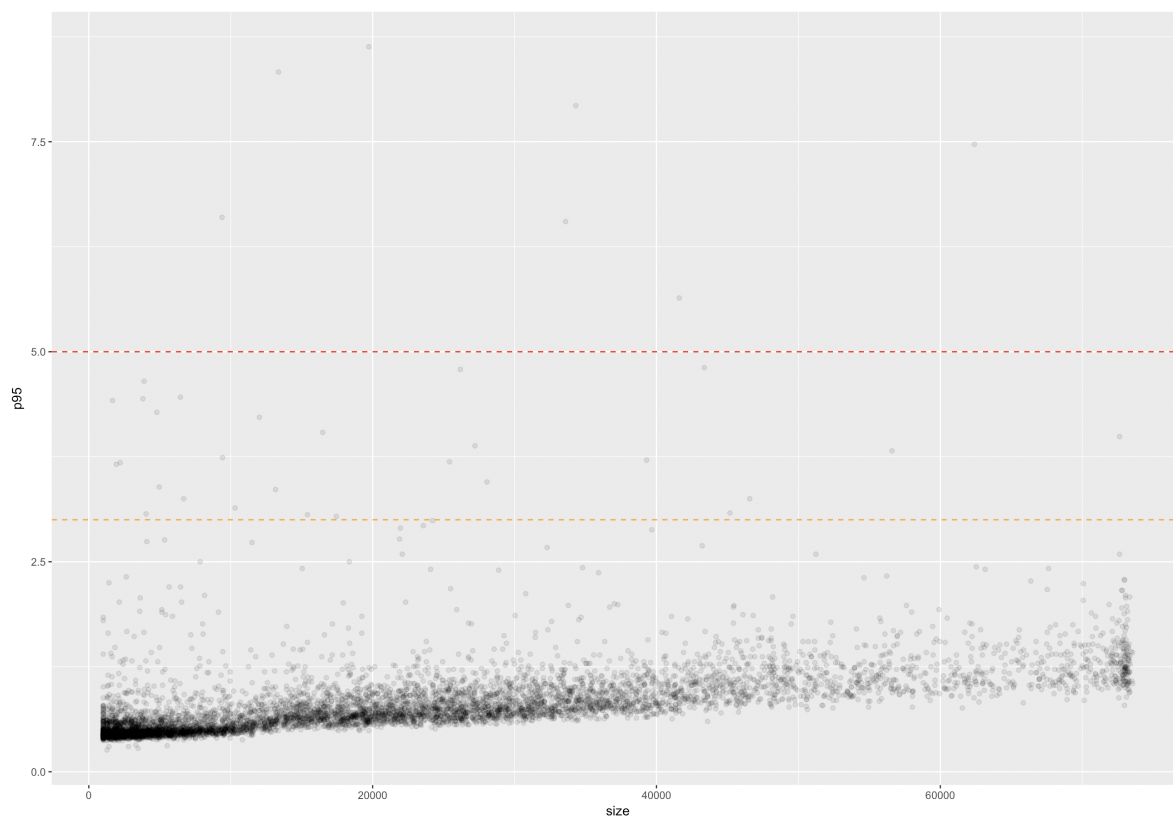



Figure 15. 95th centile of block diffusion times as a function of block size (in seconds).

Work is ongoing to further improve the performance of Cardano, based securely in the ΔQSD paradigm.

5. A Formalisation of ΔQSD

The examples that were presented in Section 4 all build on the formalisms that we will present in this section. We start by describing the notational conventions that we will use here (Section 5.1). We then provide the syntax (Definition 1) for outcome expressions and formalise the rewrite rules that define the valid transitions between possible outcomes (Definition 3). In Sections 5.3 and 5.4, we provide corresponding denotational semantics for both timeliness and load. These provide the bases for constructing formal timeliness and load analyses that can be used as part of ΔQSD. The analyses have so far been deployed manually to inform design decisions for a number of complex real-world systems. Our longer-term intention is that they should be implemented as part of a design exploration toolset that will support ΔQSD. Additional semantics and analyses are also possible, of course, and could be used to support alternative design explorations or to provide further details about timeliness, load etc.

5.1. Notational Conventions

Let $\mathbb{A}, \mathbb{B}, \mathbb{C}, \dots$ range over sets of values, and let lower case letters, $a, b, c, \dots$ range over elements of those sets. For predicates, we write $\mathrm{pred}(x)$.

5.2. Syntax

Let $\mathbb{B} \ni \flat$ and $\mathbb{O}_v \ni o_v$. We refer to black boxes and outcome variables together as *base* variables: $\overline{\mathbb{B}} = \mathbb{O}_v \cup \mathbb{B}$, where $\overline{\mathbb{B}} \ni \beta$.

Definition 1. *The abstract syntax of outcome expressions is:*

$$
\begin{aligned}
o \quad ::= \quad & \flat \mid o_v \\
\mid \quad & o \bullet\!\!\rightarrow\!\!\bullet o' \quad && \textit{sequential composition} \\
\mid \quad & o \xrightleftharpoons[m']{m} o' \quad && \textit{probabilistic choice} \\
\mid \quad & \forall(o \parallel o') \quad && \textit{all-to-finish (a.k.a. last-to-finish)} \\
\mid \quad & \exists(o \parallel o') \quad && \textit{first-to-finish}
\end{aligned}
$$

We take $o \parallel o'$ to be commutative.

In Section 4, we used these syntax elements as follows:

- $\flat$ in Section 4.1.
- o_v and $o \bullet\!\!\rightarrow\!\!\bullet o'$ throughout Section 4.
- $o \xrightleftharpoons[m']{m} o'$ in Section 4.3.
- $\forall(o \parallel o')$ in Sections 4.6 and 4.7.
- $\exists(o \parallel o')$ in Section 4.7.

Definition 2. *The* evaluation contexts $\mathcal{C}$ *of an outcome are defined as follows:*

$$
\mathcal{C} ::= [] \mid \mathcal{C} \bullet\!\!\rightarrow\!\!\bullet o \mid o \bullet\!\!\rightarrow\!\!\bullet \mathcal{C} \mid \mathcal{C} \xrightleftharpoons[m']{m} o \mid o \xrightleftharpoons[m']{m} \mathcal{C} \mid \forall(\mathcal{C} \parallel o) \mid \exists(\mathcal{C} \parallel o).
$$

where "[]" is the empty context.

Evaluation contexts are useful in the definition of *outcome transitions*, which we define next.

Definition 3. *Outcome transitions* $\tau_o : o \to o'$ *are defined by the following rewrite rules:*

$$
\begin{array}{llr}
\mathcal{C}[\flat] \to \mathcal{C}[o] & o \notin \mathbb{B} & (\textsc{Unbx}) \\
\mathcal{C}[o_v] \to \mathcal{C}[o] & o \notin \overline{\overline{\mathbb{B}}} & (\textsc{Elab}) \\
\mathcal{C}[o] \to \mathcal{C}[o' \xrightarrow{\frac{m}{m'}} o''] & \text{for some } m, m' \in \mathbb{R}^+, o', o'' \in \mathbb{O} & (\textsc{Prob}) \\
\mathcal{C}[o] \to \mathcal{C}[\forall (o' \parallel o'')] & \text{for some } o', o'' \in \mathbb{O} & (\textsc{A2F}) \\
\mathcal{C}[o] \to \mathcal{C}[\exists (o' \parallel o'')] & \text{for some } o', o'' \in \mathbb{O} & (\textsc{F2F}).
\end{array}
$$

686 *Formally speaking, a refinement step is an instance of an outcome transition. The formal description of the*
687 *system is refined when one or more refinement steps are taken.*

688   The restriction on $(\textsc{Unbx})$ is because it makes no sense to replace a black box with another black
689 box. (See the trailing discussion of Section 3.1 on the intention behind black boxes.) The restriction
690 on $(\textsc{Elab})$ is because it makes no sense for an outcome variable to be replaced by another outcome
691 variable or a black box.

692   Considering Definition 3 to be part the syntax is unusual. After all, evaluation contexts are a
693 formalism for the semantics of programming languages. However, for ΔQSD, it turns out that the
694 rewrites only cause syntactic changes to the outcome expressions (and the corresponding diagrams).
695 Note that a refinement is not a system evolution, but rather, an update in the system description. It is
696 only at analysis time that one tries to understand the *meaning* of an outcome diagram/expression.

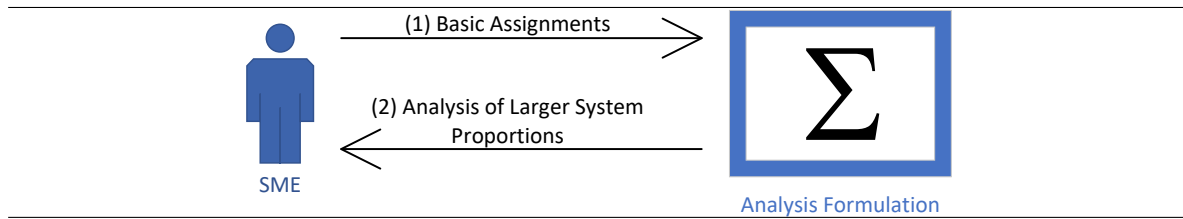



Figure 16. For an analysis, the design engineer provides basic assignments and receives more advanced values for larger parts of the system.

697 *5.3. Timeliness Analysis*

698   We are now ready to describe the process of ΔQ analysis. The idea is that the design engineer
699 provides the basic ΔQ analysis to the formulation in Definition 4. Our formulation then enables them
700 to determine the ΔQ analysis of the larger parts of their system, or even all of it. This formulation
701 is both compositional and simple. We call the ΔQ analysis that is provided by the design engineer
702 the *basic (ΔQ) assignment* (Definition 4). In the basic assignment, the design engineer only maps $\overline{\overline{\mathbb{B}}}$
703 expressions. They map those expressions to either CDFs or ΔQ variables. In return, they receive more
704 complex ΔQ expressions. This is shown in Figure 16. The process is similar for load analysis, except
705 that, there, the values exchanged between the design engineer and the respective formulation refer
706 instead to static amounts of work.

707   The reason for including the CDFs in the input type of basic assignments is rather obvious.
708 The choice to allow ΔQ variables here might be less so. The assignment of those $\overline{\overline{\mathbb{B}}}$ expressions that
709 are mapped to ΔQ variables are considered to be left by the design engineer for later. As such, the
710 formulation in Definition 4 takes the ΔQ value of those expressions to be $\top$. That is to let the design
711 engineer investigate feasibility even when those particular expressions are disregarded for the moment.

Fix a set $\Gamma \ni \gamma$ of all CDFs. Fix also a countable set of ΔQ variables $\Delta_v \ni \delta_v$. Let $\Delta = \Delta_v \cup \Gamma$, where $\Delta \ni \delta$.

Definition 4. *Given a basic assignment $\Delta_\circ[.] : \overline{\mathbb{B}} \to \Delta$, define $\Delta Q[.]_{\Delta_\circ} : \mathbb{O} \to \Gamma$ such that*

$$
\begin{aligned}
\Delta Q[\beta]_{\Delta_\circ} &= \begin{cases} \top & \text{when } \Delta_\circ[\beta] \notin \Gamma \\ \Delta_\circ[\beta] & \text{otherwise} \end{cases} \\
\Delta Q[o \bullet\!\!\rightarrow\!\!\bullet\, o']_{\Delta_\circ} &= \Delta Q[o]_{\Delta_\circ} * \Delta Q[o']_{\Delta_\circ} \\
\Delta Q[o \xleftarrow[m]{m} o']_{\Delta_\circ} &= \frac{m}{m+m'} \Delta Q[o]_{\Delta_\circ} + \frac{m'}{m+m'} \Delta Q[o']_{\Delta_\circ} \\
\Delta Q[\forall (o \parallel o')]_{\Delta_\circ} &= \max(\Delta Q[o]_{\Delta_\circ}, \Delta Q[o']_{\Delta_\circ}) \\
\Delta Q[\exists (o \parallel o')]_{\Delta_\circ} &= \min(\Delta Q[o]_{\Delta_\circ}, \Delta Q[o']_{\Delta_\circ}) \\
\text{where} \quad * \quad &\text{denotes the convolution of two } \Delta Qs
\end{aligned}
$$

We denote the set of all basic assignments by $\{\Delta_\circ[.]\}$.

We demonstrated the use of this definition in Section 4.2. In programming language theory, Definition 4 is said to give a *denotational semantics* for $\mathbb{O}$. This is because the formulation works by compositionally denoting the $\mathbb{O}$ syntax into a familiar domain, which is deemed to be simpler (Γ here), for example. Definition 4 gives the design engineer the possibility of determining the ΔQ behaviour of a *snapshot* of their system. Armed with that information, the design engineer needs to figure out whether such ΔQ behaviour is affordable. In other words, they need to make sure the *actual* ΔQ is within the acceptable bounds. In order to do that, we assume that the design engineer's customer will provide them with a *demand* CDF: one that defines the acceptable bounds. Definition 5 below is a recipe for comparing the actual behaviour against a demand CDF.

Definition 5. *Given a demand CDF γ and a partial order $<$ on Γ, say that a basic assignment $\Delta_\circ$ is a witness that an outcome o is a hazard w.r.t. γ*

$$\Delta_\circ \models_< \mathsf{hazard}_\gamma(o)$$

when

$$\Delta Q[o]_{\Delta_\circ} \nless \gamma.$$

Likewise, say $\Delta_\circ$ is a witness that an outcome o has slack once compared with γ

$$\Delta_\circ \models_< \mathsf{slack}_\gamma(o)$$

when

$$\Delta Q[o]_{\Delta_\circ} < \gamma.$$

The formulation of Definition 5 enables the design engineer to perform the ΔQ analysis of a single snapshot of their system. In some cases that is enough because it can, for example, reveal the absolute infeasibility of a design. For the majority of cases, however, it is not enough. After all, a snapshot ΔQ analysis might not be conclusive, for a variety of reasons. For example, one might not see any indication of a hazard by employing just Definition 5 because more detail is required. That takes us to Definition 8. When a design engineer works out the ΔQ analysis of a snapshot, the results might be favourable at the given level of refinement but still inaccurate. In such a case, a design engineer may wish to refine the system and perform the snapshot ΔQ again to check whether the refinement confirms the initial ΔQ analysis. Definition 8 examines that overall confirmation. Definitions 6 and 7 set the stage.

Definition 6. *Let $\Delta_\circ$ be a basic assignment. Write*

$$D_\Gamma(\Delta_\circ) = \{\beta \in \overline{\overline{\mathbb{B}}} \mid \Delta_\circ(\beta) \in \Gamma\}$$

for those $\overline{\overline{\mathbb{B}}}$ outcomes in the domain of $\Delta_\circ$ that $\Delta_\circ$ maps to CDFs.

Definition 7. *Say $\Delta_\circ'$ refines $\Delta_\circ$ (write $\Delta_\circ \to_\Delta \Delta_\circ'$) when*

- $D_\Gamma(\Delta_\circ) \subseteq D_\Gamma(\Delta_\circ')$
- $\forall \beta \in D_\Gamma(\Delta_\circ). \Delta_\circ(\beta) = \Delta_\circ'(\beta).$

In such a case, call $\Delta_\circ \to_\Delta \Delta_\circ'$ a ΔQ refinement. When clear, we will replace $\to_\Delta$ by $\to$. □

In words, a basic assignment refines another one when it keeps all the CDFs in place and possibly adds more. We are now ready for Definition 8.

Definition 8. *Fix an outcome transition $o \to o'$ and a ΔQ refinement $\Delta_\circ \to \Delta_\circ'$. Given a partial order $<$ on Γ, we say that $\Delta_\circ \to \Delta_\circ'$ witnesses that $o \to o'$ arms a hazard*

$$\Delta_\circ \to \Delta_\circ' \models_< \mathsf{hazard}(o \to o')$$

when $\Delta Q[o]_{\Delta_\circ} \not< \Delta Q[o']_{\Delta_\circ'}$. Likewise, say $\Delta_\circ \to \Delta_\circ'$ witnesses that $o \to o'$ leaves the system slack

$$\Delta_\circ \to \Delta_\circ' \models_< \mathsf{slack}(o \to o')$$

when $\Delta Q[o]_{\Delta_\circ} < \Delta Q[o']_{\Delta_\circ'}$.

As can be seen from Definitions 5 and 8, all the decisions for the timeliness analysis are made by scrutinising the CDFs (which represent ΔQ values). This is a consequence of the simple denotational semantics of Definition 4. The fact that the latter formalism is denotational implies that comparisons can be made in the domain of CDFs. Moreover, these comparisons are affordable because the denotational semantics is simple (despite being effective).

5.4. Load Analysis

This section describes how the same approach can be used to analyse the load on given resources. Resources can be of different types, in particular we distinguish *ephemeral* resources that are available at a certain rate, and *fixed* resources that are available in a fixed number or amount. Examples of ephemeral resources are CPU cycles, network interface capacity, and disk IO operations. Fixed resources include CPU cores, memory capacity and disk capacity. In this paper, we consider only ephemeral resources. The analysis that we want is an answer to the following question: *will the resource manage the amount of work assigned to it in the available time frame?*

We first need to set up some terminology for specifying the available time frame as well as the amount of work that is assigned to a given resource. Write $t^\circ(o)$ for the time an observable from the starting set of an outcome o occurs. Let $t^\infty(o) = t^\circ(o) + d(o)$, where $d(o)$ denotes the duration limit of o. Fix a set of resources $\mathbb{H} \ni \rho$. Note that the amount of work that is assigned to a resource ρ is not scalar. It is, of course, necessary to provide the unit of measurement. For example, when ρ represents CPU resources, a sensible unit of measurement is the number of CPU cycles. When ρ represents network resources, a sensible unit of measurement is the message size. At the current level of formalisation, however, we wish to set ourselves free from thinking about units of measurement. Therefore, given a resource ρ, we write W_ρ for the set of values of **the right unit of measurement** for an amount of work that has been assigned to ρ.

The design engineer utilises our load analysis in the same way that they utilise our ΔQ analysis. That is, they must provide some basic load analysis (Definition 9). Then, exactly as shown in Figure 16,

767 they use the formulation in Definition 10 to determine the load analysis for larger parts of their system,
768 or possibly all of it. We now formalise what we mean by a basic load analysis.

Definition 9. *For a given ρ, a basic "static (amount of) work assignment for ρ" is a function:*

$$\rho \left\Vert \overset{W}{\vphantom{|}}\!\!\!\!- S_\circ [.] : \overline{\mathbb{B}} \to W_\rho.$$

Definition 10. *Given a basic static work assignment $S_\circ$ for ρ, the static work assignment (i.e., the amount of work to perform a single outcome per unit of size)*

$$\rho \left\Vert \overset{W}{\vphantom{|}}\!\!\!\!- S[.]_{S_\circ}(.) : \mathbb{O} \to T \to W_\rho$$

(where T stands for time) is defined as

$$
\begin{aligned}
\rho \left\Vert \overset{W}{\vphantom{|}}\!\!\!\!- S[\beta]_{S_\circ}(t) &= \rho \left\Vert \overset{W}{\vphantom{|}}\!\!\!\!- S_\circ[\beta] & t &\in [t^\circ(o), t^\infty(o)] \\
\rho \left\Vert \overset{W}{\vphantom{|}}\!\!\!\!- S[o \bullet\!\!\rightarrow\!\!\bullet o']_{S_\circ}(t) &= \begin{cases} \rho \left\Vert \overset{W}{\vphantom{|}}\!\!\!\!- S[o]_{S_\circ}(t) \\ \rho \left\Vert \overset{W}{\vphantom{|}}\!\!\!\!- S[o']_{S_\circ}(t) \end{cases} & \begin{aligned} t &\in [t^\circ(o), t^\infty(o)) \\ t &\in [t^\circ(o'), t^\infty(o')] \end{aligned} \\
\rho \left\Vert \overset{W}{\vphantom{|}}\!\!\!\!- S[o \tfrac{m}{m'} o']_{S_\circ}(t) &= \tfrac{m}{m+m'} \times \rho \left\Vert \overset{W}{\vphantom{|}}\!\!\!\!- S[o]_{S_\circ}(t) + \tfrac{m'}{m+m'} \times \rho \left\Vert \overset{W}{\vphantom{|}}\!\!\!\!- S[o']_{S_\circ}(t) \\
& \qquad t \in [\min(t^\circ(o), t^\circ(o')), \max(t^\circ(o), t^\circ(o'))]
\end{aligned}
$$

$$
\begin{aligned}
\rho \left\Vert \overset{W}{\vphantom{|}}\!\!\!\!- S[\forall(o \parallel o')]_{S_\circ}(t) &= \\
\rho \left\Vert \overset{W}{\vphantom{|}}\!\!\!\!- S[\exists(o \parallel o')]_{S_\circ}(t) &= \rho \left\Vert \overset{W}{\vphantom{|}}\!\!\!\!- S[o]_{S_\circ}(t) + \rho \left\Vert \overset{W}{\vphantom{|}}\!\!\!\!- S[o']_{S_\circ}(t) \\
& \qquad t \in [\min(t^\circ(o), t^\circ(o')), \max(t^\circ(o), t^\circ(o'))]. \square
\end{aligned}
$$

769 Whether or not a given resource ρ is overloaded when performing an outcome o is determined by
770 whether ρ can bear the offered load in the required duration, $d(o)$. The smaller that $d(o)$ is, the faster
771 (i.e., the more *intensely*) o must be performed. However, that can only be done up to a certain threshold
772 that is determined by the system's configuration. In other words, whether the *intensity* brought to ρ
773 passes a given threshold is what determines whether ρ is overloaded. As with W_ρ, at our current level
774 of abstraction, we wish to disregard the units of measurement for intensity. That is, we write I_ρ for the
775 set of values of the right unit of measurement for the intensity of the load that is imposed on ρ. We
776 single out $\theta_I(\rho) \in I_\rho$ for the threshold of intensity ρ can bear. When it is clear, we write θ_I for $\theta_I(\rho)$.

Definition 11. *For a fixed ρ, given a threshold of intensity $\theta_I(\rho)$ and a basic static work assignment $S_\circ$ for ρ, the static slack of an outcome in ρ-consumption:*

$$S_\circ \models_\rho \mathsf{slack}_{\theta_I}(.) : \mathbb{O} \to T \to I_\rho$$

is defined as

$$S_\circ \models_\rho \mathsf{slack}_{\theta_I}(o) = \theta_I - \frac{\rho \left\Vert \overset{W}{\vphantom{|}}\!\!\!\!- S[o]_{S_\circ}}{d(o)}.$$

Define the static hazard of an outcome in ρ-consumption:

$$S_\circ \models_\rho \mathsf{hazard}_{\theta_I}(o) = -S_\circ \models_\rho \mathsf{slack}_{\theta_I}(o).$$

777 $\hfill\square$

778 Our emphasis on considering the analyses of Definitions 9–11 "static" is intentional. Firstly, they all
779 assume that a base outcome's work is spread uniformly over its duration limit. That is obviously not
780 always correct. The work assignment typically varies over the duration limit. However, if to every
781 base outcome β, the design engineer chooses to assign the highest amount of work that β needs to do

during its duration limit, the analyses given in Definition 11 would lead to a safe upper bound, that is useful as a first estimate. Secondly, Definitions 9–11 assume that an outcome's amount of work is always the same throughout its execution. Again, that is not realistic. Various reasons might cause the amount of work assigned to a base outcome to change over time. Examples are congestion, nonlinear correlations between outcomes, and cascading effects. This suggests more advanced load analyses that are "dynamic" rather than the "static" ones we have described here. We leave the development of such analyses to future work.

6. Related Work

Several theoretical or practical approaches have previously been proposed that address parts of the problem that has been identified above, but none of these addresses the whole problem in a comprehensive way.

6.1. Alternative Theoretical Approaches

6.1.1. Queuing theory

Steady-state performance has been widely studied as an aid to analysis, for example in queuing theory. Such approaches tend to take a resource-centric view of the system components, focusing on their individual utilisation/idleness. Where job/customer performance is considered, such as in mean-value analysis [19] or Jackson/BCMP networks [20], it is also in the context of steady-state averages. However, these traditional approaches cannot deliver metrics such as the time distribution of the system's response to an individual stimulus or even the probability that such a response will occur within a given time bound. These metrics are key for any time-critical and/or customer-experience-centric service.

6.1.2. Extending existing modelling approaches

With the exception of hard real-time systems, it is rare to see performance treated as a "first-class citizen" in a system design process. At best, performance is considered as a property that will emerge during the system development life-cycle, and thus something that can only be *retrospectively* validated. In contrast with ΔQSD, performance is thus *unverifiable* when using such an approach.

A common approach has been to extend existing approaches to modelling distributed systems, such as Petri nets or process calculi, with additional features with the goal of integrating performance modelling into existing design processes. Examples include stochastic Petri nets [21], timed and probabilistic process calculi [22,23] and performance evaluation process algebra (PEPA [24]). These systems consider *passage-time* [25], the time taken for the system to follow a particular path to a state, that path being characteristic of an outcome of interest [26–29]. As described above, these are all *retrospective* validation tools, requiring fully specified systems, that will give probabilistic measures of outcomes under steady-state assumptions. These systems are susceptible to state space explosion as a model grows in complexity, and this therefore limits their usage to less complex systems. Furthermore, like queuing models, they do not model *failure*, nor do they model typical real-world responses to failure such as timeouts and retries.

6.1.3. Real-time systems and Worst-case Execution Time

In real-time systems actions must be completed by strict deadlines. Missed deadlines can be catastrophic (hard real-time systems) or lead to significant delay and loss caused by roll-backs or recovery (soft real-time systems). Performance analysis has focused on giving guarantees that deadlines can be met by studying worst case execution time [30]. These approaches generally aim to analyse the behaviour of specific implementations, providing information about specific interactions. This approach is thus complementary to design-time approaches such as ΔQSD.

6.2. Distributed System Design

Designing large distributed systems is costly and error-prone. This might seem paradoxical given the proliferation of modern Internet-based companies whose core business is based on large distributed systems, such as Google, Facebook, Amazon, Twitter, Netflix and many others. Given the existence of these successful companies, it might seem that building large distributed systems is a solved problem. It is not. Successful companies have built their systems over many years, using vast amounts of effort and ingenuity to find usable solutions to difficult problems.

6.2.1. Iterative design

There does not exist a standard approach for designing large distributed systems that allows prediction of high-load performance early on during the design process. We explain the problem by giving an overview of the current design approach for distributed systems. The approach is iterative. It starts with a specification of the system's desired performance and scale. The system architecture is then designed by determining the system components according to the system's scale and estimating the performance they must have to give the required overall performance. The next step is performance validation, to verify that the design satisfies the performance requirements.

6.2.2. Testing and simulation

Performance validation is performed either as part of unit, subsystem and/or system testing, or via discrete-event simulation. Testing the performance of a component or subsystem is inconclusive without a reliable means to relate it to the resulting system performance, and testing of the whole system only reveals issues very late in the system development life-cycle. It is good practice to perform integration testing at this late stage. However, this is a poor and expensive substitute for performance analysis throughout the development process. Simulation can be performed earlier in the development process, and may be less costly than testing, but it is limited in its ability to expose rare cases, and hence cannot test tight bounds on the performance.

In the final analysis, obtaining reliable performance numbers at high load requires actually building a large part of the final system and subjecting it to a realistic load. If the system does not satisfy the requirements, then it is back to the drawing board. The system architecture is redesigned to remove observed and predicted bottlenecks and rebuilt.

6.2.3. Development risks

Several iterations of the design may be necessary until the system behaves satisfactorily. It often happens that the system only behaves satisfactorily at a fraction of the required load, but because of market constraints, this is considered acceptable and the system is deployed. In parallel to the deployment, the design engineers continue to work on a system that will accept the larger load, under the assumption that the deployment will be successful so that the load will increase.

This methodology is workable, but it is highly risky due to its high cost and development time. To have a good chance of success, it requires experienced developers. The development budget may be exhausted before achieving a satisfactory system; it may even be determined that the requirements are impossible to satisfy (infeasibility). If this is discovered early on, then the company may be able to retarget itself to become viable. Otherwise, the company simply folds.

6.2.4. Role of the ΔQSD paradigm in distributed system design

The ΔQSD paradigm is designed specifically to reduce cost and development time. The system is designed as a sequence of increasingly refined outcome diagrams. At each stage, performance is computed using the ΔQ parameters. If the system is infeasible, this is detected early on and it is immediately possible to change the design. If the design has sufficient slack, then the design process continues. The ΔQSD paradigm is effective insofar as the ΔQ computations provide realistic

results. This depends on: i) having correct ΔQ distributions for the basic components; and ii) correctly specifying causality and resource constraints. Experience with ΔQSD in past industrial designs gives us confidence in the validity of the results. The additional rigour that is provided by the ΔQSD formalism that has been introduced in this paper gives us confidence that the paradigm is being applied correctly and allows the paradigm to be integrated into new design tools.

6.3. Programming Languages and Software Engineering

6.3.1. Programming Paradigms

Programming paradigms each focus on their particular discipline for bringing more opportunities for code reuse. The most familiar examples are perhaps Object-Oriented Programming, Functional Programming, and Genericity by Type, which promote code-reuse between a base class and derived ones, by refactoring into functions, and type parameterisation. Gibbons [31] has an excellent survey on different flavours of Generic Programming with the different opportunities for code reuse that each provides. Some programming paradigms have widely-accepted formalisms and some do not. Regardless of the underlying programming paradigm, ΔQSD is a paradigm for systems development rather than simply for programming, and comes with its own formalism.

6.3.2. Software Development Paradigms

Three paradigms focus on the process of software development and are, hence, closer to ΔQSD:

Design-by-Contract. [32] Similarly to ΔQSD, in this paradigm, the programmer begins by coding by describing the pre-conditions and the post-condition. Over the years, the concept of refining initial designs from specification to code has gained increasing weight [33]. Unlike ΔQSD, the focus, however, is on functional correctness rather than performance.

Software Product Lines. [34] This paradigm targets families of software systems that are closely related and that clearly share a standalone base. The aim is to reuse the development effort of and the code for the base across all the different variations in the family. The similarity with ΔQSD is that this approach also allows variation in the implementation so long as the required quality constraints are met. In other words, variations can share a given expected outcome and its quality bounds.

Component-Based Software Engineering. [35] Components, in this paradigm, are identified by their so-called 'requires' and 'provides' interfaces. That is, so long as two components have the same 'requires' and 'provides' interfaces, they are deemed equivalent in this paradigm, and, can be used interchangeably. In ΔQSD, subsystems can also have quality contracts that involve quantitative 'demand' and 'supply' specifications. Such contracts impose quality restrictions (say, timeliness or pattern of consumption) on the respective outcomes of those subsystems. We have not shown examples of quality contracts in this paper, however, because their formalisation is not yet complete.

6.3.3. Algebraic Specification and Refinement

Algebraic specification languages such as CLEAR [36], Extended ML [37], and CASL [38] work on the basis of specifying requirements using algebraic signatures and equations that are then refined progressively until one makes it to the level of actual code. Refinement in such languages is managed using various media, for example by module systems with rigorously defined formal semantics. Whilst the focus of such languages is almost exclusively on functional correctness, studying possibilities for enhancing algebraic specifications so that they also accommodate the quality of outcomes would be an interesting avenue for future work.

914 6.3.4. Amortised Analysis

915 Amortised resource analysis is an approach for promoting resource analysis as a first-class citizen
916 of programming languages specification. Various operational semantics, type systems, and category
917 theoretical approaches have been employed. See [39], [40], and [41], for example, where memory
918 consumption for functional languages such as HASKELL and ML are automatically calculated for
919 programs written in those languages. ΔQSD advises on specification at the much higher level of
920 outcomes and outcome diagrams, leaving the actual implementation and its host language completely
921 loose. As a result, ΔQSD is much more flexible and also fruitful in swift performance workout
922 throughout the system development life-cycle.

923 **7. Conclusions**

924 This paper has presented the ΔQSD systems development process, that is steered by performance
925 predictability concerns, and that is supported by a rigorous formalism (Section 5). Our formalisation
926 of ΔQSD is a part of a wider initiative both within PNSol and IOHK [10]. ΔQSD has been successfully
927 used in a wide range of industries, including telecommunications, avionics, space and defence, and
928 cryptocurrency. It complements other approaches that are focused primarily on functional concerns,
929 such as functional programming or model checking.

930 ΔQSD is based on taking the observable *outcomes* of a system as the central point of focus
931 (Section 3.1), capturing the causal dependencies between outcomes in the form of outcome diagrams
932 (Section 3.2). The formalism also describes the process of refining outcome diagrams (Definition 7), as
933 part of a system design process. The formal specification of a system serves as a basis for different
934 analyses such as timeliness (Section 5.3) and behaviour under load (Section 5.4). Although we have
935 illustrated the ΔQSD paradigm in the context of design refinement, the aim is that these aspects should
936 permeate throughout the complete System Development Life-Cycle. Our formalism builds on the
937 simple concept of quality attenuation (ΔQ, Section 3.3) that captures the notion of performance hazard.
938 This helps early detection of infeasibility, so preventing the waste of resources (financial, people, time
939 and systems).

940 *7.1. Future work*

941 Future work will include the development of new analyses for non-ephemeral resources and for
942 dynamic loads, and an extension to non-linear systems in which the load and timeliness are coupled. In
943 parallel, we plan to use our formalism as an intermediate step to better teaching and dissemination of
944 ΔQSD. We will build tools that will enable us to understand how to track the key observables/outcomes
945 from the the design into the implementation so that they can support ongoing system design and
946 development throughout the system development life-cycle. The wider ΔQ framework is also under
947 active development within the international Broadband Forum [42] as a means of characterising quality
948 attenuation associated with networks.

949

950 1.   Narcisse, E. What Went Wrong With OnLive? Technical report, Kotaku, 2012.
951 2.   Wolverton, T. Exclusive: OnLive assets were sold off for just $4.8 million. Technical report, 2012.
952 3.   The Verge. OnLive lost: how the paradise of streaming games was undone by one man's ego. Technical
953   report, 2012.
954 4.   Pressman, R.; Maxim, D.B.R. *Software Engineering: A Practitioner's Approach*; McGraw-Hill, 2014.
955 5.   Krasner, H. The Cost of Poor Software Quality in the US: A 2020 Report. Technical report, CISQ Consortium
956   for Information & Software Quality, 2020.
957 6.   Perry, D.E.; Wolf, A.L. Foundations for the study of software architecture. *ACM SIGSOFT Software
958   Engineering Notes* **1992**, *17*, 40–52.
959 7.   Alford, M.W. A requirements engineering methodology for real-time processing requirements. *IEEE
960   Transactions on Software Engineering* **1977**, *1*, 60–69.

8.　　Solutions, P.N. Assessment of traffic management detection methods and tools. Technical Report MC-316, Ofcom, 2015.

9.　　Thompson, P.; Davies, N. Towards a RINA-Based Architecture for Performance Management of Large-Scale Distributed Systems. *Computers* **2020**, *2*.

10.　Kant, P.; Hammond, K.; Coutts, D.; Chapman, J.; Clarke, N.; Corduan, J.; Davies, N.; Díaz, J.; Güdemann, M.; Jeltsch, W.; Szamotulski, M.; Vinogradova, P. Flexible Formality Practical Experience with Agile Formal Methods. Trends in Functional Programming; Byrski, A.; Hughes, J., Eds.; Springer International Publishing: Cham, 2020; pp. 94–120.

11.　Drescher, D. *Blockchain Basics: A Non-Technical Introduction in 25 Steps*; Apress: Frankfurt am Main, Germany, 2017.

12.　David, B.; Gaži, P.; Kiayias, A.; Russell, A. Ouroboros Praos: An Adaptively-Secure, Semi-synchronous Proof-of-Stake Blockchain. Advances in Cryptology – EUROCRYPT 2018; Nielsen, J.B.; Rijmen, V., Eds.; Springer International Publishing: Cham, 2018; pp. 66–98.

13.　Coutts, D.; Davies, N.; Szamotulski, M.; Thompson, P. Introduction to the design of the Data Diffusion and Networking for Cardano Shelley. Technical report, IOHK, 2020.

14.　Watts, D.J., Small Worlds: the Dynamics of Networks between Order and Randomness; Princeton University Press, 2003; p. 280.

15.　Leon Gaixas, S.; Perello, J.; Careglio, D.; Gras, E.; Tarzan, M.; Davies, N.; Thompson, P. Assuring QoS Guarantees for Heterogeneous Services in RINA Networks with ΔQ. IEEE International Conference on Cloud Computing Technology and Science (CloudCom). IEEE, 2016, pp. 584–589.

16.　Trivedi, K.S. *Probability and statistics with reliability, queuing, and computer science applications*, 2 ed.; Wiley: New York, NY, USA, 2002.

17.　Thompson, P.; Davies, N. Towards a RINA-Based Architecture for Performance Management of Large-Scale Distributed Systems. *Computers* **2020**, *2*.

18.　Voulgaris, S. Private Communication.

19.　Reiser, M.; Lavenberg, S.S. Mean-value analysis of closed multichain queuing networks. *Journal of the ACM* **1980**, *27*, 313–322.

20.　Jackson, J.R. Jobshop-like queueing systems. *Management science* **1963**, *10*, 131–142.

21.　Molloy, M.K. Performance analysis using stochastic petri nets. *IEEE Transactions on Computers* **1982**, *31*, 913–917.

22.　Moller, F.; Tofts, C. A Temporal Calculus for Communicating Systems. CONCUR '90; Baeten, J.C.M.; Klop, J.W., Eds. Springer-Verlag, 1989, Vol. 458, *LNCS*, pp. 401–415.

23.　Jou, C.C.; Smolka, S.A. Equivalences, Congruences and Complete Axiomatizations of Probabilistic Processes. CONCUR '90; Baeten, J.C.M.; Klop, J.W., Eds. Springer-Verlag, 1989, Vol. 458, *LNCS*, pp. 367–383.

24.　Hillston, J. *A Compositional Approach to Performance Modelling*; Cambridge University Press, 1996.

25.　Bradley, J.T.; Dingle, N.J.; Gilmore, S.T.; Knottenbelt, W.J. Derivation of passage-time densities in PEPA models using IPC: The Imperial PEPA Compiler. 11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunications Systems, 2003, pp. 344–351.

26.　Daduna, H. Burke's Theorem on Passage Times in Gordon-Newell Networks. *Advances in Applied Probability* **1984**, *16*, 867–886.

27.　Guang-Hui, H.; Xue-Ming, Y. First passage times and their algorithms for markov processes. *Communications in Statistics. Stochastic Models* **1995**, *11*, 195–210, [https://doi.org/10.1080/15326349508807338]. doi:10.1080/15326349508807338.

28.　Bradley, J.; Davies, N. Performance Modelling and Synchronisation. Workingpaper, University of Bristol, United Kingdom, 1998. Other: CSTR-98-009, Superseded by CSTR-99-002.

29.　Chatrabgoun, O.; Daneshkhah, A.; Parham, G. On the functional central limit theorem for first passage time of nonlinear semi-Markov reward processes. *Communications in Statistics - Theory and Methods* **2020**, *49*, 4737–4750, [https://doi.org/10.1080/03610926.2019.1606917]. doi:10.1080/03610926.2019.1606917.

30.　Wilhelm, R.; Engblom, J.; Ermedahl, A.; Holsti, N.; Thesing, S.; Whalley, D.; Bernat, G.; Ferdinand, C.; Heckmann, R.; Mitra, T.; Mueller, F.; Puaut, I.; Puschner, P.; Staschulat, J.; Stenström, P. The Worst-Case Execution-Time Problem—Overview of Methods and Survey of Tools. *ACM Trans. Embed. Comput. Syst.* **2008**, *7*. doi:10.1145/1347375.1347389.

31. Gibbons, J., Ed. *Generic and Indexed Programming - International Spring School, SSGIP 2010, Oxford, UK, March 22-26, 2010, Revised Lectures*, Vol. 7470, *Lecture Notes in Computer Science*. Springer, 2012. doi:10.1007/978-3-642-32202-0.

32. Meyer, B. Applying "Design by Contract". *Computer* **1992**, *25*, 40–51. doi:10.1109/2.161279.

33. Weigand, H.; Dignum, V.; Meyer, J.J.C.; Dignum, F. Specification by Refinement and Agreement: Designing Agent Interaction Using Landmarks and Contracts. Engineering Societies in the Agents World III, Third International Workshop, ESAW 2002, Madrid, Spain, September 16-17, 2002, Revised Papers; Petta, P.; Tolksdorf, R.; Zambonelli, F., Eds. Springer, 2002, Vol. 2577, *Lecture Notes in Computer Science*, pp. 257–269. doi:10.1007/3-540-39173-8_19.

34. Apel, S.; Batory, D.B.; Kästner, C.; Saake, G. *Feature-Oriented Software Product Lines - Concepts and Implementation*; Springer, 2013. doi:10.1007/978-3-642-37521-7.

35. Pree, W. Component-Based Software Development - A New Paradigm in Software Engineering? *Softw. Concepts Tools* **1997**, *18*, 169–174.

36. Baumeister, H. Relating Abstract Datatypes and Z-Schemata. Recent Trends in Algebraic Development Techniques, 14th International Workshop, WADT'99, Château de Bonas, France, 1999, Selected Papers; Bert, D.; Choppy, C.; Mosses, P.D., Eds., 2000, LNCS Vol. 1827, pp. 366–382.

37. Kahrs, S.; Sannella, D.; Tarlecki, A. The Definition of Extended ML: A Gentle Introduction. *Theor. Comput. Sci.* **1997**, *173*, 445–484. doi:10.1016/S0304-3975(96)00163-6.

38. Astesiano, E.; Bidoit, M.; Kirchner, H.; Krieg-Brückner, B.; Mosses, P.D.; Sannella, D.; Tarlecki, A. CASL: The Common Algebraic Specification Language **2002**. *286*, 153–196.

39. Simões, H.R.; Vasconcelos, P.B.; Florido, M.; Jost, S.; Hammond, K. Automatic amortised analysis of dynamic memory allocation for lazy functional programs. ACM SIGPLAN International Conference on Functional Programming, ICFP'12, Copenhagen, Denmark, September 9-15, 2012; Thiemann, P.; Findler, R.B., Eds. ACM, 2012, pp. 165–176. doi:10.1145/2364527.2364575.

40. Jost, S.; Vasconcelos, P.B.; Florido, M.; Hammond, K. Type-Based Cost Analysis for Lazy Functional Languages. *J. Autom. Reason.* **2017**, *59*, 87–120. doi:10.1007/s10817-016-9398-9.

41. Rajani, V.; Gaboardi, M.; Garg, D.; Hoffmann, J. A unifying type-theory for higher-order (amortized) cost analysis. *Proc. ACM Program. Lang.* **2021**, *5*, 1–28. doi:10.1145/3434308.

42. Thompson, P.; Hernadaz, R. Quality Attenuation Measurement Architecture and Requirements. Technical Report TR-452.1, Broadband Forum, 2020.