Mind Your Outcomes

Quality-Centric Systems Development in a Pure Functional Framework

Seyed Hossein Haeri
Formal Methods Team, IOHK
Department of Informatics, University of Bergen, Norway
hossein@uib.no

Neil Davies
Predictable Network Solutions Ltd, Stonehouse, UK
neil.davies@pnsol.com

Peter Thompson
Predictable Network Solutions Ltd, Stonehouse, UK
peter.thompson@pnsol.com

Peter Van Roy
Department of Computing Science and Engineering,
Université catholique de Louvain, Belgium
pvr@info.ucl.ac.be

ABSTRACT

This paper defines the ΔQ*SD* language that embodies the main concepts of the ΔQ framework for distributed systems design. The ΔQ framework has been developed over three decades to design large distributed systems with predictable behaviour under high applied load. The framework specifies system designs at different levels of refinement and tracks and predicts their performance envelope as a function of load. System designers can thereby determine whether the system is likely to perform satisfactorily before the system is actually built. This is a critical property for real-world systems: they are expected to perform well in exactly those cases when it is difficult to ensure adequate performance, such as telephony systems during natural catastrophes.

The ΔQ*SD* language defines a system as a formal structure (called an "outcome diagram") that makes explicit how all system behaviours relate to each other. The system's design is then a sequence of refinement steps starting from a system with wholly unspecified structure and ending with a system with completely specified structure. We give the language semantics that allows computation of the predicted system performance at any step in the refinement process. In future work we intend to incorporate ΔQ*SD* in software tools and to use it to provide formal proofs that a partially specified design's performance is adequate or inadequate. This will make it a practical and useful tool for system designers.

ACM Reference Format:
Seyed Hossein Haeri, Neil Davies, Peter Thompson, and Peter Van Roy. 2021. Mind Your Outcomes: Quality-Centric Systems Development in a Pure Functional Framework. In *Proceedings of 33rd Symposium on Implementation and Application of Functional Languages (IFL 2021).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
IFL 2021, 01–03 Sep 2021, Online
© 2021 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

ACKNOWLEDGMENTS

This research was sponsored by IOHK under grant number 7006 for the Gödel project.

1 INTRODUCTION

Designing large distributed systems is a key part of network-based infrastructure and applications, but engineering these systems to have predictable performance at high applied load is very hard. However, predictability at such loads can be critical to the system's usefulness and effectiveness. For example, many of today's telecommunications systems suffer from overload when catastrophes occur, because of the large number of people placing simultaneous calls; but this is exactly when the system is in most need of predictable performance. The problem is not limited to telecommunications; in fact many large distributed systems have the same problem: distributed databases, social networks, industrial control systems, and so forth, including the Internet itself built on top of the IP best-effort packet transfer protocol. To address this problem in its generality, the ΔQ framework was developed over three decades and has been used successfully to design several large systems [3, 4].

The purpose of this paper is to define a formal language ΔQSD ("ΔQ for System Design") that embodies the main concepts of the ΔQ framework. This will allow ΔQ to be incorporated into software tools; it will help disseminate the framework by facilitating teaching materials; it will help designers use the framework to convince management of system problems (backed-up by formal proofs); and it will form a solid foundation for further development.

This paper presents the first step in a long-term research effort on formalisation of the established ΔQ framework. The ultimate goal of this work is a high-level design language for describing system designs at different levels of refinement, from specification down to implementation. A key property is that formal detection of infeasibility should be possible at any level of refinement, before hard-to-reverse decisions have been taken, and in particular, before the system is actually built. This will allow avoiding wasted effort in building systems that cannot handle the required load. The formalisation is important not only for correctness but also so that design engineers can present a convincing reasoned conclusion of infeasibility to management during the design process.

1.1 Motivation

Large distributed systems with stringent quality requirements for many users are difficult to design well. In particular, it is difficult to determine at design time how they will behave in high load situations. In many situations, design engineers simply overdimension the system as a rule of thumb. In several large companies known to the authors, engineers simply double the capacity of the system's main components with respect to a credible worst case estimate of typical conditions (in their engineering judgement). This is unsatisfactory for several reasons.

Firstly, the overdimensioning may not actually solve the problem and the system may still fail to deliver acceptable performance (or even just fail entirely) under high load. No guarantees can be given. After all, overdimensioning is just a rule of thumb – one that is implicitly assuming that the extent and duration of load saturation is strongly bounded. It is not actively managing saturation conditions.

Secondly if, during the design process, the need for a major refactoring becomes inevitable, then the engineers have no systematic way of presenting this issue (and the refinement steps previously taken) to stakeholders to convince them that corrective action is needed before the system is actually built.

Finally, from a scientific viewpoint, a more satisfactory approach would start from basic principles and allow quantitative prediction of system behaviour at high load, given only a partial specification of the system design. Such an approach creates specifications for the remaining components, permitting concurrency in the development of the design. This approach is widely used in other engineering disciplines where a system failure would be catastrophic. For example, bridge designers carefully determine the strength of a bridge's design against both static and dynamic load, using both simulation and computation, well before building the actual bridge.

The key property of physical-world design processes is that their modelling techniques have an effective notion of compositionality. The ΔQSD approach is endeavouring to achieve this for the ICT-world.

Performance is a complex concept with many axes that system designers need to consider. Out of all those axes – bandwidth, latency, scalability, and so on – there is one property that is the most important of all, namely feasibility. Feasibility is a predicate over performance: Can the system provide acceptable performance with the available resources under all specified circumstances including load levels; yes or no? This determination must be possible at design time, before the system is actually built. We call a system that is not able to satisfy this condition *infeasible*. Building an infeasible system is a huge waste of resources in money, people, and time; so, it must be avoided. The first goal of the ΔQ framework is to help the system designers detect infeasibility as early as possible. Early detection implies early circumvention and redesign, when possible. This is also the first goal of our formal language.

The reader may wonder why determining infeasibility is difficult. Let us focus for a moment on infeasibility due to delay surpassing an acceptable maximum. Would it not be possible to simply add up the delays of the system's subcomponents, and see if the resulting delay is low enough? In fact, this simple approach does not work. The problem is that system behaviour cannot easily be predicted in high load situations. Many factors contribute to this unpredictability, such as failures (transient or otherwise), congestion, jitter, translations between protocols, and outside interference. These factors interact in complex ways, which are often highly nonlinear or discontinuous. Any approach for determining infeasibility needs to take these factors into account, without overly complicating the design process. Achieving these together is nontrivial and explains why no widely used approach exists yet.

1.2 Basic Concepts of the ΔQ Framework

We first give a brief introduction to the main concepts of the ΔQ framework, on which the rest of this work is based. There are three basic concepts: outcome, quality attenuation, and hazard. We give brief definitions of these concepts below. More details and formal definitions are given in the rest of the paper.

- An *outcome* is a desired behaviour of the system. Formally, an outcome has a pair of events that denote the beginning and the end of the outcome, where an *event* is something that happens in the system at a specific location and a specific time. We can then define a service as a (usually distributed) process that generates outcomes. The specific outcomes that are important for a given system depend on the system requirements and are defined by the system designer. Some examples of outcomes are a web page download, the display of a video frame, and the response to a transaction request of a database. See Section 3.2 for more details.
- *Quality attenuation*, denoted by ΔQ, is a function giving the cumulative probability that the delay associated with an outcome is less than or equal to a given value, with respect to the value. The cumulative probability can be an improper value, i.e., when time increases indefinitely it may converge to a probability less than 1.
  Outcomes where the end event is beyond some defined delay limit can be seen as failures - this encompasses both typical notion of failure (crash etc.) as well as excessive delays.
  In this way, the ΔQ function models both delay and failure of the outcome. See Section 2 for more details.
- A *hazard* is a system effect that excessively increases quality attenuation. For example, a hazard can be armed by an internal correlation, where a part of the system interferes with another (e.g., two system components interfere when they are implemented on the same CPU core), or an external correlation such as interaction between users (e.g., congestion, where a component is overloaded by too many simultaneous requests). The framework classifies hazards [4] according to their complexity and defines mathematical models for some of those hazards (Definitions 3.5, 3.8, and 3.11). In practice, hazards affect how ΔQ computations are done for the complete system, as explained below.

1.3 Outcome Diagram

Based on these three concepts, the ΔQ framework defines an algorithm for computing system performance at design time. The system itself is represented by a formal diagram, called an *outcome diagram*, that takes into account both the system block diagram and the system outcomes that are considered important by the designer. The outcome diagram is a directed graph where nodes

are outcomes and edges are causal relationships between outcomes. The outcome diagram defines how system outcomes are related to the outcomes of its subsystems. The outcome diagram can be defined even for partially specified systems (where some outcomes are still black boxes); it requires only that undefined subsystems have well-specified behaviours.

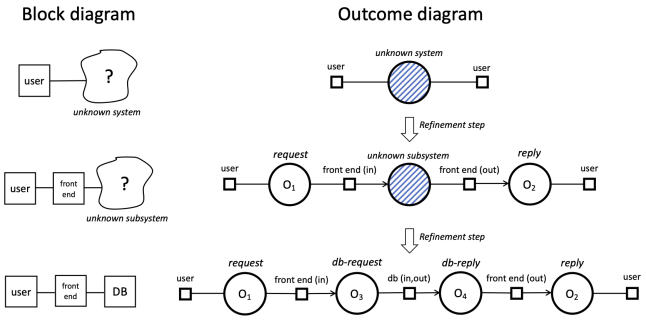

Figure 1: Block diagram and outcome diagram for a simple system constructed through stepwise refinement

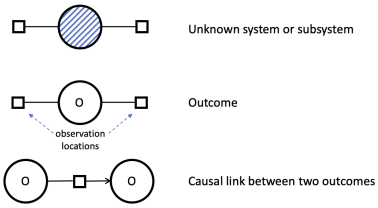

Figure 2: Legend for the outcome diagram of Figure 1

Figure 1 shows both a block diagram and outcome diagram for a simple system consisting of a user querying a front end that is connected to a database. The edges in the outcome diagram are labelled by the observation locations (small squares). Each outcome (large open circle) consists of events happening at the incoming and outgoing locations. The figure shows the refinement process of system design: a system with initially unknown structure is refined stepwise into a system with a completely known structure.

It is important to understand the fundamental difference between a block diagram and an outcome diagram. A block diagram shows the system's structure in terms of its subsystems and base components. An outcome diagram does not correspond directly to system structure. An outcome diagram is a causal graph that shows how the system's outcomes are related to each other. This is a more general concept than a block diagram. For a given outcome diagram, there can be more than one functional decomposition into a block diagram.

The outcome diagram is a key concept of the ΔQ framework. It can be used to compute the overall ΔQ of an outcome by composing the ΔQs of the finer-grain outcomes of the system, taking the relevant hazards into account. Section 2 shows how this works by giving an example of a simple system and its outcome diagram. Section 3 gives a formal definition of the outcome diagram and its semantics.

1.4 Principles of the Formal Design Language

The ΔQSD language defined in Section 3 formalises the outcome diagram and its refinement steps. The language is purely functional and its semantics is defined using a mathematical approach very similar to denotational semantics. An outcome diagram, called an outcome expression in the language, denotes a snapshot of the system design at a given level of detail. An outcome expression can contain both undefined and defined subsystems.

We define a *system design* as a sequence of outcome expressions that starts with a completely unknown system and ends with the fully specified system including all components and outcomes. Figure 1 gives an example of a system design consisting of a sequence of three outcome expressions. Each step in the sequence follows a refinement rule defined by the language. For each expression in the sequence, the language semantics gives the predicted performance as a ΔQ. An expression's semantics can be related to that of the previous and later expressions in the sequence.

By changing the sequence while respecting the refinement rules, a designer can explore the design space while satisfying performance requirements. Feasibility can be checked at any time during the design process, and if the system is infeasible the language gives a formal argument that demonstrates it. This shows the usefulness of formalising a design as a sequence of outcome expressions.

1.5 Structure of this Paper

The paper is structured into the following sections.

- Section 2 gives a more precise definition of the basic concepts of the ΔQ framework, including an example of an outcome diagram and how it is used to compute overall ΔQ.
- Section 3 contains the main contributions of the paper. It defines the abstract syntax of the ΔQSD language which defines outcome expressions and refinement steps between them. It defines the semantics of ΔQSD which allows computing ΔQ for any outcome expression, and it connects ΔQSD to other concepts in the ΔQ framework.
- Section 4 recapitulates the main contributions of the paper and outlines the further work that needs to be done.

2 QUALITY ATTENUATION

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

Because the response of the system in any particular instance can depend on a wide range of factors, including the availability of shared resources, we model ΔQ as a random variable. This allows various sources of uncertainty to be captured and modelled, ranging

from as-yet-undecided aspects of the design to resource use by other processes to dependence of behaviour on data values.

In capturing the deviation from ideal behaviour, ΔQ incorporates both delay (a continuous random variable) and exceptions/failures (discrete variables). This can be modelled mathematically using improper random variables (IRVs), whose total probability is less than one. If we write ΔQ(x) for the probability that an outcome occurs in a time $t \leq x$, then we can define the intangible mass of such an IRV as $1 - \lim_{x \to \infty} \Delta Q(x)$, which encodes the probability of exception or failure. This is illustrated by Figure 3, showing the cumulative distribution function (CDF) of an IRV[1] (with arbitrary time units).

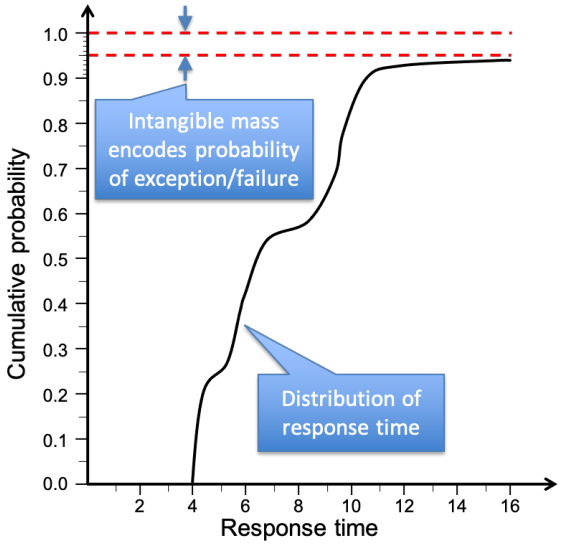

Figure 3: Cumulative distribution of an IRV

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:

$$(\forall_x \Delta Q_1(x) \leq \Delta Q_2(x)) \equiv \Delta Q_1 \geq \Delta Q_2$$

This partial order has a 'top' element, which is simply perfect performance: $\top \equiv (\forall_x \Delta Q(x) = 1)$, and a 'bottom' element, which is total failure (an outcome that never occurs): $\bot \equiv (\forall_x \Delta Q(x) = 0)$.

We can write specifications for system performance using this partial order by requiring the delivered ΔQ to be less than or equal to a predefined bounding case. Where the delivered ΔQ is strictly less than the requirement, we say there is *slack*; when it is not less than or equal to the requirement, we say there is a *hazard*. More details of this approach are given in [4].

2.1 Simple Example

Consider the commonplace distributed system of a web browser and a set of servers providing a web page. The outcome of interest to the user starts with the event of a user clicking on a URL, and end with the event of the page being fully rendered, corresponding to the first row of Figure 1. The second row of the figure shows the distinction between the user and the browser, and the third exposes the back-end servers. A typical web page will contain a variety of elements served by different domains, so for each element the browser (and its supporting O/S) must first resolve the corresponding domain name; establish a connection to the given server; and download and render the provided content. Thus for each element, the ΔQ of completing this process is the sequential composition of the ΔQs of the component steps; and the ΔQ of rendering the whole page is a last-to-finish combination of the ΔQs of all the elements. Note that this formulation *automatically* deals with the possibility that any of the steps may fail, and provides the resultant failure probability for the whole process in addition to the distribution of expected completion time.

We can further refine the model: for example the DNS resolution of a domain may provide alternative server addresses for load-balancing, and these servers may have different ΔQs for providing the content. We can represent this as a probabilistic choice between these ΔQs, weighted by the chances of the respective servers being provided.

We can additionally consider the effect of load and contention for shared resources, for example network interface bandwidth or rendering capacity, or the impact of different DNS caching architectures. These aspects of system performance design are formalised below.

3 FORMALISM

In remainder of this text, we take the *system* of discourse to be fixed for the *Subject Matter Expert* (SME). We assume their system has a number of tasks to perform. In order to perform a task, the system might need to perform several other subtasks, or a task might be considered atomic by the SME. The process of looking into more details of the system by breaking tasks into their pertaining subtasks is what we call *refinement*. (See Definition 3.3 for the formal definition.) By refining a system, one goes from a coarser *granularity* to a finer one. In that vein, a quality-centric system design is a sequence of consecutive refinements from the topmost outcome to the lowest level of granularity.

3.1 Notational Conventions

Subscripts and priming do not change the syntactic category of a symbol. For example, for a set A, we write $A \ni a$ to indicate that $a, a', a'', \ldots, a_1, a_2, \ldots$ all range over A.

We employ fonts to communicate information about the syntactic categories. Let x be an object of interest. When referring to the set of all such objects, we write $\mathbb{X}$. For predicates, we write $\text{pred}(x)$.

3.2 Outcomes

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

[1] In this paper we regard and IRV as equivalent to its CDF.

Description of system performance in terms of the outcomes expected from it is a novelty of our work. In this section, we formalise different aspects of such a description, the result of which we call the *outcome expression* of a system. (See Definition 3.1.) The outcome expression is a basis for different analyses we put forward. As will become clear in Section 3.3 and Section 3.4, an outcome expression is like the syntax for a variety of semantices à la polymorphic embedding [1].

Two distinct sets of events are attributed to each outcome: the starting set and the terminating set. 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 of an outcome o is of interest because it signifies the point in time at which as well as the 3D location – altogether referred to as the 4D location or simply location – at which o's performance began. Likewise, an observable from the terminating set of o is an event with information regarding its location. Of course, once an observable from o's starting set occurs, there is no guarantee that one from o's terminating set will also occur within o's *duration limit*. But, when an observable from o's terminating set occurs within the duration limit after an observable from its starting set occurred, o is said to be *done*.

Diagrammatically, we show an outcome using a circle. In an *outcome diagram*, we depict the starting set and the terminating set of an outcome using small boxes to the left and to the right of the outcome's circle, respectively. We connect the starting set to the outcome from the left and to the terminating set from the right. Fig. 1 presents all that. When unimportant for an outcome, we do not include the starting set and the terminating set of that particular outcome in the outcome diagram. In terms of graph theory, observables are optional labels for the edges. We maintain a directional convention to avoid showing directions explicitly for every edge: When an edge connects two outcomes, the one depicted to the right causally depends on the one depicted to the left.

We single out two sets of outcomes.

Consider the situation where an SME is aware that an outcome is not atomic; nevertheless, they take a level of refinement enough for carrying out a particular analysis. (See Section 3.3 and Section 3.4 for two possible analyses.) In our formalism, the description of that intention at that level of refinement employs a *black box* for that particular outcome. As such, black boxes $\mathbb{B} \ni \flat$ are those outcomes that:

- can be easily quantified;
- are beyond the SME's control (and so may need to be quantified by external specification or measurement);
- the SME intentionally leaves the details for later.

Outcome variables $\mathbb{O}_v \ni o_v$ are the variables that we use for referring to a given outcome. 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 3.1. Define the abstract syntax of outcome expressions as:

o	::=	$\flat \mid o_v$	
	|	$o \bullet\!\!\rightarrow\!\!\bullet\, o'$	sequential composition
	|	$o \overset{m}{\underset{m'}{\rightleftharpoons}} o'$	probabilistic choice
	|	$\forall(o \parallel o')$	all-to-finish (a.k.a. last-to-finish)
	|	$\exists(o \parallel o')$	first-to-finish

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

A probabilistic choice $o \overset{m}{\underset{m'}{\rightleftharpoons}} o'$ is the same as o with the probability $\frac{m}{m+m'}$ and same as o' with the probability $\frac{m'}{m+m'}$. For two outcomes o and o' started at the same time and run in parallel, $\forall(o \parallel o')$ is done when both o and o' are done. Similarly, $\exists(o \parallel o')$ is done as soon as the first of o and o' is done.

Definition 3.2. The evaluation contexts C of an outcome are defined as follows:

$$C ::= [] \mid C \bullet\!\!\rightarrow\!\!\bullet\, o \mid o \bullet\!\!\rightarrow\!\!\bullet\, C \mid C \overset{m}{\underset{m'}{\rightleftharpoons}} o \mid o \overset{m}{\underset{m'}{\rightleftharpoons}} C \mid \forall(C \parallel o) \mid \exists(C \parallel o).$$

where "[]" is the empty context. □

The evaluation contexts become handy in the definition of outcome transitions, which we define next.

Definition 3.3. Outcome transitions $\tau_o : o \to o'$ are defined by the following rewrite rules:

$C[\flat] \to C[o]$	$o \notin \mathbb{B}$	(Unbx)
$C[o_v] \to C[o]$	$o \notin \overline{\mathbb{B}}$	(Elab)
$C[o] \to C[o' \overset{m}{\underset{m'}{\rightleftharpoons}} o'']$	for some $m, m' \in \mathbb{R}^+$, $o', o'' \in \mathbb{O}$	(Prob)
$C[o] \to C[\forall(o' \parallel o'')]$	for some $o', o'' \in \mathbb{O}$	(A2F)
$C[o] \to C[\exists(o' \parallel o'')]$	for some $o', o'' \in \mathbb{O}$	(F2F).

Formally speaking, a refinement step is an instance of an outcome transition. The formal description of the system is refined upon taking one or more refinement steps. □

Note that a refinement is not a system evolution. A refinement is an update in the system description.

3.3 ΔQ Analysis

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 $\Delta \ni \delta$. We are now ready to describe the process of ΔQ analysis.

The idea is that the SME gives the formulation we are about to provide (Definition 3.4) the basic ΔQ analysis. Based on that analysis, our formulation enables them to work out the ΔQ analysis of the larger parts of their system, or even all of it. The formulation is compositional and simple.

We call the ΔQ analysis that the SME provides the *basic (ΔQ) assignment*. In the basic assignment, the SME only maps $\overline{\mathbb{B}}$ expressions. And, they map those expressions to either CDFs or ΔQ variables.

The reason for the inclusion of the CDFs is rather obvious. The choice to allow ΔQ variables here might be less so. The assignment of those $\overline{\mathbb{B}}$ expressions that are mapped to ΔQ variables are considered to be left by the SME for later. As such, the formulation in Definition 3.4 takes the ΔQ value of those expressions to be $\top$. That is to let the SME to investigate feasibility even when those particular expressions are disregarded for the moment.

Definition 3.4. Given a basic assignment $\Delta_\circ[\![.]\!] : \overline{\mathbb{B}} \to \Delta$, define $\Delta Q[\![.]\!]_{\Delta_\circ} : \mathbb{O} \to \Gamma$ such that

$$\begin{aligned}
\Delta Q[\![\beta]\!]_{\Delta_\circ} &= \begin{cases} \top & \text{when } \Delta_\circ[\![\beta]\!] \notin \Gamma \\ \Delta_\circ[\![\beta]\!] & \text{otherwise} \end{cases} \\
\Delta Q[\![o \bullet\!\!\to\!\!\bullet\, o']\!]_{\Delta_\circ} &= \Delta Q[\![o]\!]_{\Delta_\circ} \oplus \Delta Q[\![o']\!]_{\Delta_\circ} \\
\Delta Q[\![o \underset{m}{\overset{m}{\leftrightarrows}} o']\!]_{\Delta_\circ} &= \tfrac{m}{m+m'} \Delta Q[\![o]\!]_{\Delta_\circ} + \tfrac{m'}{m+m'} \Delta Q[\![o']\!]_{\Delta_\circ} \\
\Delta Q[\![\forall(o \parallel o')]\!]_{\Delta_\circ} &= \max(\Delta Q[\![o]\!]_{\Delta_\circ}, \Delta Q[\![o']\!]_{\Delta_\circ}) \\
\Delta Q[\![\exists(o \parallel o')]\!]_{\Delta_\circ} &= \min(\Delta Q[\![o]\!]_{\Delta_\circ}, \Delta Q[\![o']\!]_{\Delta_\circ})
\end{aligned}$$

Denote the set of all basic assignments by $\{\Delta_\circ[\![.]\!]\}$. □

The above formulation gives the SME the possibility of working out the ΔQ behaviour of a **snapshot** of their system. Armed with that, the SME 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. To that end, we assume that the SME's customer provides them with a *demand* CDF: one that defines the acceptable bounds. Definition 3.5 below is a recipe for comparing the actual behaviour against a demand CDF.

Definition 3.5. 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

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

when

$$\Delta Q[\![o]\!]_{\Delta_\circ} \not< \gamma.$$

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

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

when

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

□

The formulation of Definition 3.5 enables the SME to perform the ΔQ analysis of a single snapshot of their system. In some cases, that is enough because it can, for example, reveal absolute infeasibility. For the majority of cases, however, that 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 3.5 because more detail is required. That takes us to Definition 3.8.

When an SME 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, an SME may wish to refine the system and perform the snapshot ΔQ again to check whether the refinement confirms the initial ΔQ analysis. Definition 3.8 examines that confirmation. Definitions 3.6 and 3.7 set the stage.

Definition 3.6. Let $\Delta_\circ$ be a basic assignment. Write

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

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

Definition 3.7. 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 3.8.

Definition 3.8. Fix an outcome transition $o \to o'$ and a ΔQ refinement $\Delta_\circ \to \Delta'_\circ$. Given a partial order $<$ on Γ, say $\Delta_\circ \to \Delta'_\circ$ witnesses that $o \to o'$ arms a hazard

$$\Delta_\circ \to \Delta'_\circ \models_< \text{hazard}(o \to o')$$

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

$$\Delta_\circ \to \Delta'_\circ \models_< \text{slack}(o \to o')$$

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

3.4 Load Analysis

This section aims at analysing 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 we are after in this paper is working out an answer to the following question: Will the resource manage the amount of work assigned to it in the available time frame?

To that end, we first need to set up terminology for specifying the available time frame as well as the amount of work assigned to a given resource. The next four paragraphs serve that purpose.

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[2] $\mathbb{H} \ni \rho$.

The amount of work assigned to a resource ρ is not scalar. The unit of measurement is necessary. For example, when ρ is CPU, a sensible unit of measurement is the number of CPU cycles. When ρ is network, a sensible unit of measurement is the message size.

At its 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 that an amount of work shed to ρ can take.

The SME utilises our load analysis in the same way they utilise our ΔQ analysis. That is, it is on them to provide some basic load analysis. Then, they use the formulation we are about to present (Definition 3.10) to work out the load analysis for larger parts of their system, or possibly all of it. We now formalise what we mean by a basic load analysis.

Definition 3.9. For a given ρ, a basic "static (amount of) work assignment for ρ" is a function:

$$\rho \Big\|\overset{W}{\vdash} S_\circ[\![.]\!] : \overline{\mathbb{B}} \to W_\rho.$$

Definition 3.10. Given a basic static work assignment $S_\circ$ for ρ, the static work assignment (i.e., the amount of work for a single

[2]The sensible notation for the set of all resources would have been $\mathbb{R}$, which is already reserved for real numbers. So, we chose $\mathbb{H}$ because the second letter in "rho" is 'h.'

performance of an outcome per unit of size)

$$\rho \Big|\!\!\Big|^{W} S[\![.]\!]_{S_\circ}(.) : \mathbb{O} \to T \to W_\rho$$

(where T is for time) is defined as

$$\begin{aligned}
&\rho \Big|\!\!\Big|^{W} S[\![\beta]\!]_{S_\circ}(t) = \rho \Big|\!\!\Big|^{W} S_\circ[\![\beta]\!] && t \in [t^\circ(o), t^\infty(o)]\\
&\rho \Big|\!\!\Big|^{W} S[\![o \bullet\!\!\to\!\!\bullet\, o']\!]_{S_\circ}(t) =\\
&\qquad \begin{cases} \rho \Big|\!\!\Big|^{W} S[\![o]\!]_{S_\circ}(t) & t \in [t^\circ(o), t^\infty(o))\\ \rho \Big|\!\!\Big|^{W} S[\![o']\!]_{S_\circ}(t) & t \in [t^\circ(o'), t^\infty(o')] \end{cases}\\
&\rho \Big|\!\!\Big|^{W} S[\![o \overset{m}{\underset{m'}{\rightleftharpoons}} o']\!]_{S_\circ}(t) =\\
&\qquad \tfrac{m}{m+m'} \times \rho \Big|\!\!\Big|^{W} S[\![o]\!]_{S_\circ}(t) + \tfrac{m'}{m+m'} \times \rho \Big|\!\!\Big|^{W} S[\![o']\!]_{S_\circ}(t)\\
&\rho \Big|\!\!\Big|^{W} S[\![\forall(o \parallel o')]\!]_{S_\circ}(t) =\\
&\rho \Big|\!\!\Big|^{W} S[\![\exists(o \parallel o')]\!]_{S_\circ}(t) =\\
&\qquad \rho \Big|\!\!\Big|^{W} S[\![o]\!]_{S_\circ}(t) + \rho \Big|\!\!\Big|^{W} S[\![o']\!]_{S_\circ}(t). \square
\end{aligned}$$

Whether or not a given resource ρ is overloaded upon the performance of an outcome o is determined by whether ρ can bear the offered load in $d(o)$. The smaller $d(o)$, the quicker o is to be performed, i.e., the more intensely. But, that can only be done up to a certain threshold that is determined by the system's configuration. In other words, whether the *intensity* brought to ρ passes a given threshold is what determines whether ρ is overloaded. Similar to W_ρ, at our current level of abstraction, we would like to disregard the units of measurement for intensity. That is, we write I_ρ for the set of values of the right unit of measurement the intensity of the load shed on ρ takes. We single out $\theta_I(\rho) \in I_\rho$ for the threshold of intensity ρ can bear. When clear, we write θ_I for $\theta_I(\rho)$.

Definition 3.11. 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

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

Define the static hazard of an outcome in ρ-consumption

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

□

Our emphasis on considering the analyses of Definitions 3.9–3.11 "static" is intentional.

Firstly, they all assume that a base outcome's work is spread uniformly over its duration limit. That is obviously not always correct. The work assignment typically varies over the duration limit. However, if to every base outcome β, the SME chooses to assign the highest amount of work β needs to do during its duration limit, the analyses given in Definition 3.11 would lead to a safe upperbound that is useful as a first estimate.

Secondly, Definitions 3.9–3.11 assume that an outcome's amount of work is always the same throughout the runtime. 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.

The above two paragraphs suggest more advanced load analyses that are rather "dynamic." We leave the development of such analyses to future work.

4 CONCLUSIONS

This work presents a formalism (Section 3) for a systems development process steered by performance predictability concerns. It can be seen as a complement to system development approaches which are steered by functional concerns.

The formalism is based on a vision (Section 1.1) that takes outcomes of a system as the central point of focus (Section 3.2) and captures the causal dependencies between outcomes as outcome diagrams (Section 1.3). The formalism also describes the process of refining outcome diagrams to one another (Definition 3.7), hence, modelling the steps of a system design process. The formal specification serves as a basis for different analyses of a system such as timeliness (Section 3.3) and behaviour under load (Section 3.4). Although illustrated in the context of design refinement, the aim is that these aspects should permeate throughout the complete system life-cycle. The formalism employed for those analyses builds on the simple concept of ΔQs for quality attenuation (Section 2) that enables early detection of infeasibility. As such it helps early circumvention or redesign, when possible; hence, preventing foreseeable but often neglected wasting of resources.

The formalism we present in this work is an attempt to give a well-established practice, which we call ΔQSD, a sound footing. Prior to this work, ΔQSD used concepts from pure functional languages (Haskell, in particular) by formulating the representation of the performance characteristics as, for example, monoids. ΔQSD has been successfully used in a wide range of industries, including telecommunications, avionics, space and defence, and cryptocurrency. Our formalisation of ΔQSD is a part of a wider initiative both within PNSol and IOHK [2].

Load analysis for other resources than ephemeral ones is future work. Dynamic load analysis is also an important future work for those being more realistic. In parallel, we plan to use our formalism as an intermediate step to better teaching and disseminating ΔQSD. We will build tools for understanding how to track the key observables/outcomes from the the design into the implementation so that they can support ongoing system development throughout the life-cycle. The wider ΔQ framework is also under active development in the Broadband Forum [5] as a means of characterising quality attenuation associated with networks.

REFERENCES

[1] C. Hofer, K. Ostermann, T. Rendel, and A. Moors. Polymorphic embedding of DSLs. In Y. Smaragdakis and J. G. Siek, editors, *Generative Programming and Component Engineering, 7th International Conference, GPCE 2008, Nashville, TN, USA, October 19-23, 2008, Proceedings*, pages 137–148. ACM, 2008.

[2] P. Kant, K. Hammond, D. Coutts, J. Chapman, N. Clarke, J. Corduan, N. Davies, J. Díaz, M. Güdemann, W. Jeltsch, M. Szamotulski, and P. Vinogradova. Flexible formality practical experience with agile formal methods. In A. Byrski and J. Hughes, editors, *Trends in Functional Programming*, pages 94–120, Cham, 2020. Springer International Publishing.

[3] P. Thompson and N. Davies. Towards a performance management architecture for large-scale distributed systems using rina. In *2020 23rd Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*, pages 29–34, 2020.

[4] P. Thompson and N. Davies. Towards a RINA-based architecture for performance management of large-scale distributed systems. *Computers*, 2(53), June 2020.

[5] P. Thompson and R. Hernadaz. Quality attenuation measurement architecture and requirements. Technical Report TR-452.1, Broadband Forum, September 2020.