Probabilistic Analysis of the Quality Calculus

Hanne Riis Nielson and Flemming Nielson

DTU Compute, Technical University of Denmark, Denmark
{hrni,fnie}@dtu.dk

Abstract. We consider a fragment of the Quality Calculus, previously introduced for defensive programming of software components such that it becomes natural to plan for default behaviour in case the ideal behaviour fails due to unreliable communication.

This paper develops a probabilistically based *trust* analysis supporting the Quality Calculus. It uses information about the probabilities that expected input will be absent in order to determine the trustworthiness of the data used for controlling the distributed system; the main challenge is to take accord of the stochastic dependency between some of the inputs. This takes the form of a relational static analysis dealing with quantitative information.

1 Introduction

Motivation. Distributed systems often need to continue operating in a meaningful way even when communication links become unreliable. This is particularly important in embedded systems, control systems and systems providing critical infrastructure services. Such systems form key components of cyber physical systems where the consequences of software malfunction may be disastrous. As an example, if the engine control system of a car breaks down when the car is driving at high speed, a crash might result.

Unreliability of communication links due to security attacks is relatively well understood. There are numerous cryptographic protocols for ensuring the confidentiality, integrity and authenticity of data communicated between components of distributed systems. Also there are many state-of-the-art tools for finding and eliminating security flaws in such protocols. As an example, forcing a car to brake because the tyre sensors incorrectly report loss of pressure, can be avoided using proper cryptographic protocols (although it would add to the cost of the pressure sensors and transmitters).

Unreliability of communication links due to faults and denial of service attacks is much harder to guard against. In a cyber physical system it seems hard to avoid that the communication link can be flooded with irrelevant messages, that there might be radio interference on the wireless frequency band, or that physical antennas are shielded from receiving or transmitting their signals. This calls for utilising programming notations that help to ensure that software systems are hardened against such attacks on communication. As an example, the braking

D. Beyer and M. Boreale (Eds.): FMOODS/FORTE 2013, LNCS 7892, pp. 258–272, 2013.

system of a car should be designed such that some braking effect continues to be applied even if no signals are received from the braking sensors about the spinning (and potential blocking) of the wheels.

We consider here the Quality Calculus [7], that is a recent proposal for how to enforce robustness considerations on software components executing in an open and error prone environment: what should the behaviour be when communication links may have broken down. While this is a first step in this important direction it is necessary to develop analyses to indicate the trust that we can have in the overall robustness of the system.

In this paper we develop a novel analysis for supporting the Quality Calculus. It is a probabilistically based *trust* analysis that uses information about the probabilities that expected input will be absent in order to determine the trustworthiness of the data used for controlling the distributed system; the main challenge is to take accord of the stochastic dependency between some of the probabilities determined in the availability analysis.

Overview. A fragment of the Quality Calculus [7] is reviewed in Section 2. Its distinguishing feature is a *binder* specifying the inputs to be performed before continuing. In the simplest case it is an input guard $t^{\ell}?x$ describing that some value should be received over the channel t and should be bound to the variable x; data over t is assumed to have trustworthiness as given by an element ℓ from some finite lattice of trust values. Increasing in complexity there are binders of the form $\&_q(t_1^{\ell_1}?x_1, \cdots, t_n^{\ell_n}?x_n)$ indicating that several inputs are simultaneously active and with a quality predicate q that determines when sufficient inputs have been received to continue. As a consequence, when continuing with the process after the binder, some variables might not have obtained proper values as the corresponding inputs have not been performed. To model this the calculus distinguishes between data and optional data, much like the use of option data types in programming languages like Standard ML. The construct $\mathsf{case}\ x\ \mathsf{of}\ \mathsf{some}(y) : P_1\ \mathsf{else}\ P_2$ will evaluate the variable x; if it evaluates to $\mathsf{some}(c)$ we will execute P_1 with y bound to c; if it evaluates to none we will execute P_2.

The main motivating example, an intelligent smart meter, is discussed in Section 3. It is a scenario inspired by [8] where a smart meter of a household communicates with a service provider to obtain a schedule for operating a number of appliances taking pricing and availability of energy into account.

In Section 4 we show how to make use of information about the probabilities that expected input will be absent. We develop a probabilistic trust analysis associating probability distributions with all program points of interest where the probabilities indicate the trust level of the optional data. This helps in pinpointing those parts of the code where there is a high risk of basing decisions on less trustworthy data (like default data). The main challenge is to adequately model when the probabilities are stochastically dependent and when they are not; in the parlance of static analysis this means that we need to perform relational rather than independent attribute analyses of the Quality Calculus [6]

Table 1. A fragment of the Quality Calculus

$$
\begin{array}{rcl}
P & ::= & (\nu c^{\ell})\, P \;\mid\; P_1 \,|\, P_2 \;\mid\; \mathsf{0} \;\mid\; b.P \;\mid\; t_1!t_2.P \;\mid\; A \\
& \mid & \mathsf{case}\ x\ \mathsf{of}\ \mathsf{some}(y) : P_1\ \mathsf{else}\ P_2 \\
b & ::= & \&_q(t_1^{\ell_1}?x_1, \cdots, t_n^{\ell_n}?x_n) \\
t & ::= & y \;\mid\; c \;\mid\; g(t_1, \cdots, t_n)
\end{array}
$$

and our development constitutes an extension of relational analyses to deal with quantitative information. We conclude and present our outlook on future work in Section 5.

2 Review of the Quality Calculus

A main purpose of process calculi is to focus on programming abstractions that may provide insight into the development of distributed systems. The challenge of concurrency was addressed in early process calculi like CCS [4] and the π-calculus [5] whereas the challenge of Service Oriented Computation have been addressed in calculi such as COWS [3], SOCK [2]and CaSPiS [1]. We consider here a fragment of the Quality Calculus [7] that addresses the challenge of how to ensure robustness of software systems that execute in an open environment, that does not always live up to expectations — possibly because anticipated communications do not take place due to faults or denial of service attacks.

Syntax. We now develop the syntax and semantics of a fragment of the Quality Calculus by adapting the development of [7]. A *system* consists a number of process definitions and a main process:

$$\mathsf{define}\ A_1 \triangleq P_1; \cdots; A_n \triangleq P_n\ \mathsf{in}\ P_*\ \mathsf{using}\ c_1^{\ell_1}, \cdots, c_m^{\ell_m}$$

Here A_i is the name of a process, P_i is its body, P_* is the main process and $c_1, \cdots, c_m$ is a list of constants. The syntax of processes is given in Table 1. A *process* can have the form $(\nu c^{\ell})\, P$ introducing a new constant c and its scope P, it can be a parallel composition $P_1 \,|\, P_2$ of two processes P_1 and P_2 and it can be an empty process denoted $\mathsf{0}$. An input process is written $b.P$ where b is a binder (explained below) specifying the inputs to be performed before continuing with P. An output process has the form $t_1!t_2.P$ specifying that the value t_2 should be communicated over the channel t_1. A process can also be a call A to one of the defined processes or a case construct (explained below). We shall feel free to dispense with trailing occurrences of the process $\mathsf{0}$.

To indicate the trust level of data we use an element ℓ from a finite trust lattice $\mathcal{L}$ and we write $\leq$ for the ordering on $\mathcal{L}$. As an example, we might use $\mathcal{L} = (\{\text{L}, \text{M}, \text{H}\}, \leq)$ to denote that available data is classified into low (L), medium (M) or high (H) trust and with the obvious linear ordering on these elements. More refined examples may be obtained by adopting the lattices of Mandatory

Access Control Policies or the Decentralised Label Model. All constants are annotated with an element from the trust lattice when they are introduced into the system (either in the list $c_1^{\ell_1}, \cdots, c_m^{\ell_m}$ or using the syntax $(\nu c^{\ell})\, P$); these annotations are performed by the programmers as part of a code review for determining the quality of the code.

The main distinguishing feature of the Quality Calculus is the *binder* b specifying the inputs to be performed before continuing. In the simplest case it is an input guard $t^{\ell}?x$ describing that some value should be received over the channel t and it will be bound to the variable x; the trust element ℓ indicates the amount of trust to be placed in data received over this channel. Increasing in complexity we may have binders of the form $\&_q(t_1^{\ell_1}?x_1, \cdots, t_n^{\ell_n}?x_n)$ indicating that n inputs are simultaneously active and a *quality predicate* q determines when sufficient inputs have been received to continue.

As an example, q can be $\exists$ meaning that one input is required, or it can be $\forall$ meaning that all inputs are required; formally $\exists(x_1, \cdots, x_n) \Leftrightarrow x_1 \vee \cdots \vee x_n$ and $\forall(x_1, \cdots, x_n) \Leftrightarrow x_1 \wedge \cdots \wedge x_n$. For more expressiveness we shall allow to write e.g. $[0 \wedge (1 \vee 2)]$ for the quality predicate defined by $[0 \wedge (1 \vee 2)](x_0, x_1, x_2) \Leftrightarrow x_0 \wedge (x_1 \vee x_2)$; this is particularly useful because unlike [7] we do not allow to nest binders.

As a consequence, when continuing with the process P in $b.P$ some variables might not have obtained proper values as the corresponding inputs might not have been performed. To model this we distinguish between *data* and *optional data*, much like the use of option data types in programming languages like Standard ML. In the syntax we use constants c, functions g (taking data as arguments and producing data as results), variables y and terms t to denote data and we use variables x to denote optional data; in particular, the expression $\mathsf{some}(t)$ signals the presence of some data t and none the absence of data. Returning to the processes, the construct $\mathsf{case}\ x\ \mathsf{of}\ \mathsf{some}(y) : P_1\ \mathsf{else}\ P_2$ will test whether x evaluates to some data and if so, bind it to y and continue with P_1 and otherwise continue with P_2.

We need to impose a few well-formedness constraints on systems. Names u are divided into data constants c, data variables y, and optional data variables x. For a system of the form displayed above we shall require that the main process P_* as well as the bodies P_i have no free variables (over data or over optional data) and that their free constants are among $c_1, \cdots, c_m$.

Semantics. The semantics consists of a structural congruence and a transition relation [5]. The *structural congruence* $P_1 \equiv P_2$ is defined in Table 2 and expresses when two processes, P_1 and P_2, are congruent to each other. It enforces that processes constitute a monoid with respect to parallel composition and the empty process and it takes care of the unfolding of calls of named processes and scopes for constants; here $\mathsf{fc}(P)$ is the set of constants occurring free in P. Finally, it allows replacement in contexts C given by:

$$C ::= [\,] \;\mid\; (\nu c^{\ell})\, C \;\mid\; C \,|\, P \;\mid\; P \,|\, C$$

Table 2. Structural congruence of the Quality Calculus

$P \equiv P$	$P_1 \equiv P_2 \Rightarrow P_2 \equiv P_1$
$P_1 \equiv P_2 \wedge P_2 \equiv P_3 \Rightarrow P_1 \equiv P_3$	$P \,\vert\, \mathsf{0} \equiv P$
$P_1 \,\vert\, P_2 \equiv P_2 \,\vert\, P_1$	$P_1 \,\vert\, (P_2 \,\vert\, P_3) \equiv (P_1 \,\vert\, P_2) \,\vert\, P_3$
$(\nu c^{\ell}) \, P \equiv P \quad \text{if } c \notin \mathsf{fc}(P)$	$(\nu c_1^{\ell_1}) \, (\nu c_2^{\ell_2}) \, P \equiv (\nu c_2^{\ell_2}) \, (\nu c_1^{\ell_1}) \, P \quad \text{if } c_1 \neq c_2$
$A \equiv P \quad \text{if } A \triangleq P$	$(\nu c^{\ell}) \, (P_1 \mid P_2) \equiv ((\nu c^{\ell}) \, P_1) \mid P_2 \quad \text{if } c \notin \mathsf{fc}(P_2)$
$P_1 \equiv P_2 \Rightarrow C[P_1] \equiv C[P_2]$	

The *transition relation*

$$P \longrightarrow P'$$

describes when a process P evaluates into another process P'. It is parameterised on a relation $t \rhd c$ describing when a (closed) term t evaluates to a constant c; the definition of this relation is straightforward and hence omitted. Furthermore, we shall make use of two auxiliary relations

$$c_1!c_2 \vdash b \to b'$$

for specifying the effect on the binder b of matching the output $c_1!c_2$, and

$$b ::_v \theta$$

for recording (in $v \in \{\mathsf{tt}, \mathsf{ff}\}$) whether or not all required inputs of b have been performed as well as information about the composite substitution (θ) that has been constructed. To formalise this we extend the syntax of binders to include substitutions as well as individual inputs

$$b ::= \&_q(b'_1, \cdots, b'_n) \qquad \text{with} \qquad b' ::= [\mathsf{some}(c)/x] \;\mid t^{\ell}?x$$

where $[\mathsf{some}(c)/x]$ is the substitution that maps x to $\mathsf{some}(c)$ and leaves all other variables unchanged. We shall write id for the identity substitution and $\theta_2\theta_1$ for the composition of two substitutions, so $(\theta_2\theta_1)(x) = \theta_2(\theta_1(x))$ for all x.

The first part of Table 3 defines the transition relation $P \longrightarrow P'$. The first clause expresses that the original binder is replaced by a new binder recording the output just performed; this transition is only possible when $b ::_{\mathsf{ff}} \theta$ holds, meaning that more inputs are required before proceeding with the continuation P_2. The second clause considers the case where no further inputs are required; this is expressed by the premise $b ::_{\mathsf{tt}} \theta$. In this case the binding is performed by applying the substitution θ to the continuation process — hence no further inputs are allowed. The next clauses are straightforward; they define the semantics of the case construct, how the structural congruence is embedded in the transition relation and how transitions take place in contexts.

The next clause in Table 3 defines the auxiliary relation $c_1!c_2 \vdash b \to b'$; here the idea is simply to record the binding of the value received in the appropriate position.

Table 3. Transition rules of the Quality Calculus

$$\frac{t_1 \rhd c_1 \quad t_2 \rhd c_2 \quad c_1!c_2 \vdash b \to b' \quad b' ::_{\mathsf{ff}} \theta}{t_1!t_2.P_1 \mid b.P_2 \longrightarrow P_1 \mid b'.P_2}$$

$$\frac{t_1 \rhd c_1 \quad t_2 \rhd c_2 \quad c_1!c_2 \vdash b \to b' \quad b' ::_{\mathsf{tt}} \theta}{t_1!t_2.P_1 \mid b.P_2 \longrightarrow P_1 \mid P_2\theta}$$

$$\mathsf{case\ some}(c)\ \mathsf{of\ some}(y) : P_1\ \mathsf{else}\ P_2 \longrightarrow P_1[c/y]$$

$$\mathsf{case\ none\ of\ some}(y) : P_1\ \mathsf{else}\ P_2 \longrightarrow P_2$$

$$\frac{P_1 \equiv P_2 \quad P_2 \longrightarrow P_3 \quad P_3 \equiv P_4}{P_1 \longrightarrow P_4} \qquad \frac{P_1 \longrightarrow P_2}{C[P_1] \longrightarrow C[P_2]}$$

$$\frac{t \rhd c_1}{c_1!c_2 \vdash \&_q(b_1, \cdots, t^{\ell}?x, \cdots, b_n) \to \&_q(b_1, \cdots [\mathsf{some}(c_2)/x], \cdots, b_n)}$$

$$t^{\ell}?x ::_{\mathsf{ff}} [\mathsf{none}/x] \qquad [\mathsf{some}(c)/x] ::_{\mathsf{tt}} [\mathsf{some}(c)/x]$$

$$\frac{b_1 ::_{v_1} \theta_1 \quad \cdots \quad b_n ::_{v_n} \theta_n}{\&_q(b_1, \cdots, b_n) ::_v \theta_n \cdots \theta_1} \text{ where } v = q(v_1, \cdots, v_n)$$

The auxiliary relation $b ::_v \theta$ is defined in the final group of clauses in Table 3. Here we perform a pass over the (extended) syntax of the binder b, evaluating whether or not a sufficient number of inputs have been performed and recorded in v, and computing the associated composite substitution θ.

3 Motivating Example

We now introduce our main motivating example for the analyses to be developed subsequently. It is a scenario inspired by [8] where a smart meter SM of a household communicates with a service provider SP to obtain a schedule for operating a number of appliances taking pricing and availability of energy into account. Unfortunately, the service provider is subject to denial of service attacks and therefore the smart meter is equipped with a local computer LC for computing a schedule to be used whenever the service provider does not produce a schedule. The schedule computed by the service provider is the preferred one so the smart meter will use a clock CL to set a waiting time; only if no schedule is received

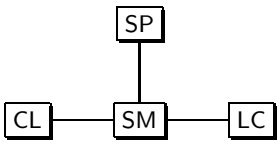

Fig. 1. The Smart Meter scenario

$$
\begin{array}{lll}
\textsf{define} & \textsf{SP} \triangleq \cdots \text{ (see text) } \cdots \\
& \textsf{LC} \triangleq \cdots \text{ (see text) } \cdots \\
& \textsf{CL} \triangleq \cdots \text{ (see text) } \cdots \\
& \textsf{SM} \triangleq \cdots \text{ (see text) } \cdots \\
\textsf{in} & \textsf{SP} \mid \textsf{LC} \mid \textsf{Clock} \mid \textsf{SM} \\
\textsf{using} & \textsf{request}^{\mathrm{H}}, \textsf{req}^{\mathrm{H}}, \textsf{rep}^{\mathrm{L}}, \textsf{set}^{\mathrm{H}}, \textsf{tick}^{\mathrm{H}}, \textsf{install}^{\mathrm{H}}, \checkmark^{\mathrm{H}}
\end{array}
$$

Fig. 2. The Smart Meter system

from the service provider within the waiting time, the smart meter will use the locally computed schedule. The overall scenario is illustrated in Figures 1 and 2 and we shall now describe the individual processes in more detail; we shall feel free to use a polyadic version of the calculus when it eases the presentation and also to use basic actions delay and wait for indicating that time passes (as explained below) but otherwise having no effect in the semantics.

The service provider SP is defined by:

$$
\begin{array}{l}
\textsf{SP} \triangleq \&_{\exists}(\textsf{request}^{\mathrm{H}}?(x_i x_r)).\ \textsf{case}\ x_i\ \textsf{of}\ \textsf{some}(y_i)\colon \\
\qquad\qquad\qquad\qquad\qquad\qquad \textsf{case}\ x_r\ \textsf{of}\ \textsf{some}(y_r)\colon \textsf{delay}_g.y_i!(\textsf{global}(y_r)).\textsf{SP} \\
\qquad\qquad\qquad\qquad\qquad\qquad \textsf{else SP} \\
\qquad\qquad\qquad\qquad\quad \textsf{else SP}
\end{array}
$$

The service provider first obtains a request of x_r resources from a smart meter identifying itself as x_i; the annotation H indicates that the channel request carries information of high trust level. The next three lines express that SP after some delay computes a schedule using the function global (taking data as input and returning data) and sends it to the smart meter before recursing. The case constructs ensure that the proper data is extracted and supplied to the function; indeed the two else branches are not reachable.

The local computer LC is similar to the service provider except that it makes use of the channels req and rep for obtaining the request and returning the reply:

$$\textsf{LC} \triangleq \&_{\exists}(\textsf{req}^{\mathrm{H}}?x_r).\ \textsf{case}\ x_r\ \textsf{of}\ \textsf{some}(y_r)\colon \textsf{delay}_l.\textsf{rep!local}(y_r).\textsf{LC else LC}$$

It uses the function local (taking data as input and returning data) to compute the schedule; once more the case construct is used to extract the actual request and the else branch is in fact not reachable.

As already mentioned the smart meter will put a limit on how long time it will wait for a schedule and the process CL below models a simple clock communicating over the channels set and tick:

$$\textsf{CL} \triangleq \&_{\exists}(\textsf{set}^{\mathrm{H}}?x_t).\ \textsf{case}\ x_t\ \textsf{of}\ \textsf{some}(y_t)\colon \textsf{wait}(c).\textsf{tick!}\checkmark.\textsf{CL else CL}$$

The idea is that the input on set starts the clock and the output of the constant ✓ on the channel tick indicates that the prescribed waiting time c has passed.

Finally, let us consider the main control process SM for the smart meter:

$$\begin{aligned}\mathsf{SM} \triangleq (\nu i_g^{\mathrm{M}})\,(\nu d^{\mathrm{H}})\ & \mathsf{request!}(i_g d).\ \mathsf{req!}d.\ \mathsf{set!}\checkmark.\\ & \&_{[0\wedge(1\vee 2)]}(\mathsf{tick}^{\mathrm{H}}?x_t, i_g^{\mathrm{M}}?x_g, \mathsf{rep}^{\mathrm{L}}?x_l).\\ & \mathsf{case}\ x_g\ \mathsf{of}\ \mathsf{some}(y_g)\colon {}^{1}\mathsf{install!}y_g.\mathsf{SM}\\ & \mathsf{else\ case}\ x_l\ \mathsf{of}\ \mathsf{some}(y_l)\colon {}^{2}\mathsf{install!}y_l.\mathsf{SM}\\ & \qquad \mathsf{else}\ {}^{3}0\end{aligned}$$

For later reference we have added labels to three subprocesses. The first line issues two requests for a schedule, one to the service provider and one to the local computer and it also starts the clock. The binder of the second line expresses that the time must have passed and that at least one schedule must have been received before continuing. Note that it is indeed possible for both schedules to arrive before the time has passed and it is also possible that no schedule has arrived when the time has passed in which case the smart meter continues waiting. The third line will give priority to the global schedule (the process labelled 1) whereas in the absence of a global schedule the fourth line will install the local schedule before recursing (the process labelled 2). As in the previous examples the case constructs are used to extract the required data and in fact the final else branch (labelled 3) is not reachable.

Discussion. As an alternative we might use the binder

$$\&_{[0\vee(1\wedge 2)]}(\mathsf{tick}^{\mathrm{H}}?x_t, i_g^{\mathrm{M}}?x_g, \mathsf{rep}^{\mathrm{L}}?x_l)$$

in line 2. Then we will stop waiting for schedules when the time bound has been met but in the case where both schedules have been received it is possible to continue before the specified time has passed. Changing the binder to

$$\&_{[0\vee(1\vee 2)]}(\mathsf{tick}^{\mathrm{H}}?x_t, i_g^{\mathrm{M}}?x_g, \mathsf{rep}^{\mathrm{L}}?x_l)$$

means that we proceed as soon as one of the schedules has arrived or the time has passed; as we shall see later this is the least attractive of the three models as concerns the robustness of the smart meter.

4 Trust Analysis

The Quality Calculus allows to describe which data is needed before continuing but given the expressiveness of the quality predicates one cannot be sure what data has actually been received. The subsequent process is able to determine this by testing for whether given data has actually been received and decisions can be made accordingly. Some of these decisions may have high trustworthiness whereas others may have low trustworthiness. In order to evaluate the overall system it is therefore important

- to be able to *annotate* the binders with information about the probability distribution over the trustworthiness of data actually received, and
- to be able to *propagate* these trust annotations throughout the process.

Trust Annotations. To annotate the binders we change the syntax of binders from $\&_q(t_1^{\ell_1}?x_1, \cdots, t_n^{\ell_n}?x_n)$ to $\&_q^\pi(t_1^{\ell_1}?x_1, \cdots, t_n^{\ell_n}?x_n)$. Here

$$\pi \in \mathcal{D}(\{x_1, \cdots, x_n\} \to \mathcal{L}_\perp)$$

is a probability distribution indicating the probability of the various inputs having been received where $\perp$ denotes the absence of input and $\mathcal{L}_\perp$ is the lifted trust lattice obtained from $\mathcal{L}$ by adding $\perp$ as the new least element. Note that the distribution assigns probabilities to tuples of inputs rather than to individual inputs because the quality predicates may be such that the various inputs are not stochastically independent — this is where our development will constitute a generalisation of relational static analyses to take care of quantitative information (in this case the probabilities).

One way to obtain the probability distribution is to run a given process for a while and to sample the actual distribution of inputs at the binders. Another way is to assume that the delays are governed by memoryless processes started just before the binders and then to use stochastic techniques to estimate the distributions. We shall illustrate the latter approach through a few examples.

Example 1. Let us consider the process SM of Section 3 and let us assume that the processes computing the global and local schedules are exponentially distributed with rates λ_g and λ_l, respectively. We want to determine the distribution π that should be used in the following binder:

$$\&^\pi_{[0\wedge(1\vee 2)]}(\mathsf{tick}^{\mathrm{H}}?x_t, i_g^{\mathrm{M}}?x_g, \mathsf{rep}^{\mathrm{L}}?x_l)$$

Let us assume that $\lambda_g = 0.2$ and $\lambda_l = 0.5$ and that the waiting time (c) is 5 time units. One can show that

$$\begin{aligned}\pi([x_t \mapsto \mathrm{H},\ x_g \mapsto \mathrm{M},\ x_l \mapsto \mathrm{L}]) &= 0.580\\ \pi([x_t \mapsto \mathrm{H},\ x_g \mapsto \mathrm{M},\ x_l \mapsto \perp]) &= 0.061\\ \pi([x_t \mapsto \mathrm{H},\ x_g \mapsto \perp,\ x_l \mapsto \mathrm{L}]) &= 0.359\end{aligned}$$

and $\pi(\sigma) = 0$ in all other cases.

Example 2. Let us next consider the variant of SM using the binder

$$\&^\pi_{[0\vee(1\wedge 2)]}(\mathsf{tick}^{\mathrm{H}}?x_t, i_g^{\mathrm{M}}?x_g, \mathsf{rep}^{\mathrm{L}}?x_l)$$

and let us, as before, assume that the processes computing the global and local schedules are exponentially distributed with rates λ_g and λ_l, respectively. Taking $\lambda_g = 0.2$ and $\lambda_l = 0.5$ we get the following distribution when the waiting time is 5 time units:

$$\begin{aligned}\pi([x_t \mapsto \perp,\ x_g \mapsto \mathrm{M},\ x_l \mapsto \mathrm{L}]) &= 0.580\\ \pi([x_t \mapsto \mathrm{H},\ x_g \mapsto \mathrm{M},\ x_l \mapsto \perp]) &= 0.052\\ \pi([x_t \mapsto \mathrm{H},\ x_g \mapsto \perp,\ x_l \mapsto \mathrm{L}]) &= 0.338\\ \pi([x_t \mapsto \mathrm{H},\ x_g \mapsto \perp,\ x_l \mapsto \perp]) &= 0.030\end{aligned}$$

and $\pi(\sigma) = 0$ in all other cases. As an example $\pi([x_t \mapsto \mathrm{H}, x_g \mapsto \mathrm{M}, x_l \mapsto \mathrm{L}]) = 0$ because it would correspond to the event that one of x_g and x_l is received before

Table 4. Trust Analysis of the Quality Calculus

$$\vdash 1, \pi_* \mathbin{@} P_* \qquad \vdash 1, \pi_* \mathbin{@} P_1 \qquad \cdots \qquad \vdash 1, \pi_* \mathbin{@} P_n$$

$$\frac{\vdash p, \pi \mathbin{@} (\nu c^{\ell})\, P}{\vdash p, (\pi|^{\mathsf{C}}_{\{c\}}) \otimes \delta_{[c \mapsto \ell]} \mathbin{@} P} \qquad \frac{\vdash p, \pi \mathbin{@} (P_1 \mid P_2)}{\vdash p, \pi \mathbin{@} P_1} \qquad \frac{\vdash p, \pi \mathbin{@} (P_1 \mid P_2)}{\vdash p, \pi \mathbin{@} P_2}$$

$$\frac{\vdash p, \pi \mathbin{@} (b.P) \quad \vdash b \blacktriangleright \pi_b}{\vdash p, (\pi|^{\mathsf{C}}_{\mathsf{bv}(b)}) \otimes \pi_b \mathbin{@} P} \qquad \frac{\vdash p, \pi \mathbin{@} (t_1!t_2.P)}{\vdash p, \pi \mathbin{@} P}$$

$$\frac{\vdash p, \pi \mathbin{@} (\mathsf{case}\ x\ \mathsf{of}\ \mathsf{some}(y){:}\ P_1\ \mathsf{else}\ P_2)}{\vdash p \cdot \pi_{[x \neq \bot]}, \pi_{\downarrow [x \neq \bot]}[y := x] \mathbin{@} P_1} \quad \text{if } \pi_{[x \neq \bot]} \neq 0$$

$$\frac{\vdash p, \pi \mathbin{@} (\mathsf{case}\ e\ \mathsf{of}\ \mathsf{some}(y){:}\ P_1\ \mathsf{else}\ P_2)}{\vdash p \cdot \pi_{[x = \bot]}, \pi_{\downarrow [x = \bot]} \mathbin{@} P_2} \quad \text{if } \pi_{[x = \bot]} \neq 0$$

time has passed and the other is received at the exact moment that time passes; while this is a possible event it occurs with probability 0.

Example 3. Changing the binder to

$$\&^{\pi}_{[0 \vee (1 \vee 2)]}(\mathsf{tick}^{\mathrm{H}}?x_t, i^{\mathrm{M}}_g?x_g, \mathsf{rep}^{\mathrm{L}}?x_l)$$

gives rise to the following distribution using the same parameters as above:

$$\begin{aligned} \pi([x_t \mapsto \bot,\ x_g \mapsto \mathrm{M},\ x_l \mapsto \bot]) &= 0.277 \\ \pi([x_t \mapsto \bot,\ x_g \mapsto \bot,\ x_l \mapsto \mathrm{L}]) &= 0.693 \\ \pi([x_t \mapsto \mathrm{H},\ x_g \mapsto \bot,\ x_l \mapsto \bot]) &= 0.030 \end{aligned}$$

and $\pi(\sigma) = 0$ in all other cases.

We shall find it helpful to use the notation $\vdash \&^{\pi}_{q}(t^{\ell_1}_1?x_1, \cdots, t^{\ell_n}_n?x_n) \blacktriangleright \pi$ in order to extract the probability annotation from the binder.

Trust Propagation. We now consider how to propagate the trust information from the binders throughout the entire system. The judgements of our analysis of processes P take the form

$$\vdash p, \pi \mathbin{@} P$$

Here p is the probability that we will reach this occurrence of the process P and π is a distribution from $\mathcal{D}(V \to \mathcal{L}_{\bot})$ where the free names of P are contained in the set V (determined by the detailed definition of the analysis). The mappings of $V \to \mathcal{L}_{\bot}$ assign trust levels to the free names of P and the idea is that π specifies the distribution of these mappings when P is reached. The mappings $\sigma : V \to \mathcal{L}_{\bot}$ will ensure that constants c and variables over data y only take values in $\mathcal{L}$.

A mapping $\sigma : V \to \mathcal{L}_\perp$ gives rise to distribution δ_σ in $\mathcal{D}(V \to \mathcal{L}_\perp)$, called the *Dirac distribution*, by assigning it the probability 1 and all other mappings the probability 0:

$$\delta_\sigma(\sigma') = \begin{cases} 1 \text{ if } \sigma = \sigma' \\ 0 \text{ otherwise} \end{cases}$$

We shall need a number of operations on $\mathcal{D}(V \to \mathcal{L}_\perp)$. The first is a *projection* on a subset $U \subseteq V$ of names, written $\pi\,|_U$. Given a distribution π in $\mathcal{D}(V \to \mathcal{L}_\perp)$ it will return a distribution $\pi\,|_U$ in $\mathcal{D}(U \to \mathcal{L}_\perp)$ and is defined by

$$(\pi\,|_U)(\sigma) = \Sigma_{(\sigma' \text{ s.t. } \sigma = \sigma'|_U)} \pi(\sigma')$$

Here $\sigma'\,|_U$ is the restriction of the mapping $\sigma' : V \to \mathcal{L}_\perp$ to the domain U and we thus summarise all the probabilities associated with mappings that are equal when projected on U. We shall write $\pi|_U^{\complement}$ as a shorthand for $\pi\,|_{V \setminus U}$, that is for projection on the complement of U:

$$(\pi|_U^{\complement})(\sigma) = \Sigma_{(\sigma' \text{ s.t. } \sigma = \sigma'|_{V \setminus U})} \pi(\sigma')$$

In Table 4 we shall use this operator when binding new constants in $(\nu c^\ell)\, P$ and new variables in $b.P$ — the point being that old instances of these name will no longer be in scope.

The next operation is a *product* operation; it takes two distributions $\pi_1 : \mathcal{D}(V_1 \to \mathcal{L}_\perp)$ and $\pi_2 : \mathcal{D}(V_2 \to \mathcal{L}_\perp)$ defined over disjunct sets of names (so $V_1 \cap V_2 = \emptyset$) as arguments and constructs a distribution $\pi_1 \otimes \pi_2$ in $\mathcal{D}((V_1 \cup V_2) \to \mathcal{L}_\perp)$. It is given by

$$(\pi_1 \otimes \pi_2)(\sigma) = \pi_1(\sigma\,|_{V_1}) \cdot \pi_2(\sigma\,|_{V_2})$$

Thus we split the mapping into two parts and multiply the probabilities obtained from the two distributions. We shall use this operation when combining two stochastically independent distributions. For the construct $(\nu c^\ell)\, P$ we take the product with the Dirac distribution $\delta_{[c \mapsto \ell]}$ and in the case of the input binder $b.P$ we take the product with the distribution π_b obtained from the binder b using the notation $\vdash \&_q^\pi(t_1^{\ell_1}?x_1, \cdots, t_n^{\ell_n}?x_n) \blacktriangleright \pi$.

The next operation is a simple *lookup* operation: given a distribution $\pi : \mathcal{D}(V \to \mathcal{L}_\perp)$, a name u and a trust level ℓ we shall be interested in the probability $\pi_{[u=\ell]}$ for u having the trust level ℓ:

$$\pi_{[u=\ell]} = \Sigma_{(\sigma \text{ s.t. } \sigma(u)=\ell)} \pi(\sigma)$$

In a similar way we can define $\pi_{[u \neq \ell]}$ as the probability that u does not have the trust level ℓ:

$$\pi_{[u \neq \ell]} = \Sigma_{(\sigma \text{ s.t. } \sigma(u) \neq \ell)} \pi(\sigma)$$

It is easy to see that $\pi_{[u \neq \ell]} = 1 - \pi_{[u=\ell]}$. These operations are used in the analysis of the construct $\mathsf{case}\ x\ \mathsf{of}\ \mathsf{some}(y)\colon P_1\ \mathsf{else}\ P_2$; here $\pi_{[x \neq \perp]}$ is the probability that the first branch is taken and similarly $\pi_{[x=\perp]}$ is the probability that the second branch is taken.

We shall also introduce a *selection* operation that given a distribution $\pi : \mathcal{D}(V \to \mathcal{L}_\perp)$, a name u and a trust level ℓ will construct a new distribution $\pi_{\downarrow[u \neq \ell]}$ of $\mathcal{D}(V \to \mathcal{L}_\perp)$ that gives 0 for all mappings σ with $\sigma(u) = \ell$ and rescales the remaining probabilities. The operation is only defined when $\pi_{[u \neq \ell]} \neq 0$:

$$(\pi_{\downarrow[u \neq \ell]})(\sigma) = \begin{cases} \frac{\pi(\sigma)}{\pi_{[u \neq \ell]}} & \text{if } \sigma(u) \neq \ell \\ 0 & \text{if } \sigma(u) = \ell \end{cases}$$

In a similar way we can define the operation $\pi_{\downarrow[u=\ell]}$ that gives 0 for all mappings σ with $\sigma(u) \neq \ell$ and rescales the remaining probabilities. This operation is only defined when $\pi_{[u=\ell]} \neq 0$:

$$(\pi_{\downarrow[u=\ell]})(\sigma) = \begin{cases} 0 & \text{if } \sigma(u) \neq \ell \\ \frac{\pi(\sigma)}{\pi_{[u=\ell]}} & \text{if } \sigma(u) = \ell \end{cases}$$

These operations are used for the construct $\mathsf{case}\ x\ \mathsf{of}\ \mathsf{some}(y) : P_1\ \mathsf{else}\ P_2$. Here the distribution for the first branch will be $\pi_{\downarrow[x \neq \perp]}$ assuming that it might be taken, that is, that $\pi_{[x \neq \perp]} \neq 0$. The distribution for the second branch is $\pi_{\downarrow[x=\perp]}$, again under the assumption that it might be taken, that is, that $\pi_{[x=\perp]} \neq 0$.

Finally, we need an operation for the *extension* of a distribution $\pi : \mathcal{D}(V \to \mathcal{L}_\perp)$ with a new name $u' \notin V$ that is stochastically dependent on a name $u \in V$ in the sense that u and u' have the same trust level. The resulting distribution $\pi[u' := u]$ is in $\mathcal{D}((V \cup \{u'\}) \to \mathcal{L}_\perp)$ and it is defined by

$$(\pi[u' := u])\sigma = \begin{cases} \pi(\sigma|^{\complement}_{\{u'\}}) & \text{if } \sigma(u) = \sigma(u') \\ 0 & \text{otherwise} \end{cases}$$

This operation is used in the analysis of the case construct where the distribution for the first branch needs to be extended with information about the variable y being bound by the construct so the distribution will be $(\pi_{\downarrow[x \neq \perp]})[y := x]$.

We now have the machinery needed for the definition of $\vdash p, \pi \mathbin{@} P$ in Table 4. We analyse the main process and all bodies with the initial choice of $p = 1$ and $\pi = \pi_*$ where the list of constants $c_1^{\ell_1}, \cdots, c_m^{\ell_m}$ in the system definition in Section 2 gives rise to defining $\pi_* = \delta_{[c_1 \mapsto \ell_1, \cdots, c_m \mapsto \ell_m]}$. The choice of $p = 1$ reflects that all probabilities of reaching program points are conditional on the main process being called as well as the bodies. The remaining clauses were explained when introducing the auxiliary notation.

The analysis has been implemented using Standard ML. For the examples shown here it suffices to represent distributions π as lists of pairs (σ, p) where σ is a mapping and p is a non-zero probability. A number of improvements are possible. We may use the fact that variables in the domain of distributions are often stochastically independent (as when introduced in different binders) and hence distributions could be represented as products of distributions over disjoint domains. Also we may use symbolic techniques (like Binary Decision Diagrams) to represent the distributions that cannot be split into the product of distributions over simpler domains.

Querying the Analysis. We shall give two examples of how information about outputs can be extracted from the analysis.

First let us consider an output of the form $c!t$ and assume we are interested in the trust levels of the values being communicated over the channel c. Furthermore let us assume that the analysis gives us a probability p and a distribution π satisfying $\vdash p, \pi \mathbin{@} c!t.P$. This means that the program point of interest is reached with probability p and furthermore, the distribution will be π. The trust levels of the values being communicated over c can be represented as a distribution ξ_π in $\mathcal{D}(\mathcal{L})$ and it can be defined by:

$$\xi_\pi(\ell) = \Sigma_{(\sigma \text{ s.t. } \sigma(t)=\ell)} \pi(\sigma)$$

Here $\sigma(t)$ is the trust level of the term t; for constants and variable this is straightforward as the information is available in σ whereas for terms $g(t_1, \cdots, t_n)$ we need to specify how to compute the resulting trust level from the trust levels of the arguments. So we shall assume that we have interpretations

$$\langle\!\langle g \rangle\!\rangle : \mathcal{L} \times \cdots \times \mathcal{L} \to \mathcal{L}$$

specifying this; as an example we might take $\langle\!\langle g \rangle\!\rangle(\ell_1, \cdots, \ell_n) = \ell_1 \sqcap \cdots \sqcap \ell_n$ for $\sqcap$ being the greatest lower bound of the trust lattice $\mathcal{L}$ and reflecting that a result is no more trustworthy than any of the arguments used to compute it.

Next let us consider all outputs of the form $c!\cdot$ and let us make two simplifying assumptions. The first is that the constant c is always explicit (i.e. does not arise in the form of some $y!\cdot$ where y is eventually replaced by c). The other is that all occurrences of $c!\cdot$ occur in different branches of the case constructs and that in particular no occurrence of $c!\cdot$ prefixes another. We may then calculate the distribution Ξ_c in $\mathcal{D}(\mathcal{L}_\perp)$ of the trust levels of data communicated over c. For a trust level $\ell \in \mathcal{L}$ we define

$$\Xi_c(\ell) = \Sigma_{\vdash p,\pi \mathbin{@} c!t.P} \big(\Sigma_{\sigma \text{ s.t. } \sigma(t)=\ell} p \cdot \pi(\sigma) \big)$$

where the first sum is over all occurrences of $\vdash p, \pi \mathbin{@} c!t.P$ in the analysis of the overall system of interest. For the trust level $\perp$ we define

$$\Xi_c(\perp) = 1 - \Sigma_{\ell \in \mathcal{L}} \Xi_c(\ell)$$

so as to reflect the probability that no communication over c is taking place.

Analysing the Motivating Example. We have used our implementation to analyse the smart meter example in various scenarios.

Let us first consider Example 1 where the binder used in the definition of SM is $\&^{\pi}_{[0\wedge(1\vee 2)]}(\mathsf{tick}^{\mathrm{H}}?x_t, i_g^{\mathrm{M}}?x_g, \mathsf{rep}^{\mathrm{L}}?x_l)$ and where the processes computing the schedules are exponentially distributed with rates $\lambda_g = 0.2$ and $\lambda_l = 0.5$. This corresponds to a scenario where the service provider is somewhat slower to deliver a schedule than the local computer.

Using the trust analysis we can then compute the probability Ξ_{install} of the smart meter installing a schedule computed by the service provider, a locally

computed schedule or no schedule at all; this corresponds to combining the analysis results for the program points labelled 1, 2, and 3 in the process SM in Section 3. The result is $\Xi_{\mathsf{install}}(\mathrm{H}) = 0$, $\Xi_{\mathsf{install}}(\mathrm{M}) = 0.64$, $\Xi_{\mathsf{install}}(\mathrm{L}) = 0.36$, and $\Xi_{\mathsf{install}}(\bot) = 0$. This shows that we are always sure to get a schedule installed and in 64% of the cases it is the global schedule and in the remaining 36% it is the local schedule. The information about $\Xi_{\mathsf{install}}(\mathrm{H})$, $\Xi_{\mathsf{install}}(\mathrm{M})$, and $\Xi_{\mathsf{install}}(\mathrm{L})$ originate from the points labelled 1 and 2 in the code for SM in Section 3, whereas the information about $\Xi_{\mathsf{install}}(\bot)$ originates from the point labelled 3.

Let us next consider Example 2 where the binder used in the definition of SM is $\&^{\pi}_{[0\vee(1\wedge 2)]}(\mathsf{tick}^{\mathrm{H}}?x_t, i_g^{\mathrm{M}}?x_g, \mathsf{rep}^{\mathrm{L}}?x_l)$ and let us repeat the experiments using the same parameters as above. The result is $\Xi_{\mathsf{install}}(\mathrm{H}) = 0$, $\Xi_{\mathsf{install}}(\mathrm{M}) = 0.63$, $\Xi_{\mathsf{install}}(\mathrm{L}) = 0.34$, and $\Xi_{\mathsf{install}}(\bot) = 0.03$. This shows that we can no longer be sure to get a schedule installed but that the global schedule is still much more likely than the local schedule.

Finally, let us consider Example 3 where the binder used in the definition of SM is $\&^{\pi}_{[0\vee(1\vee 2)]}(\mathsf{tick}^{\mathrm{H}}?x_t, i_g^{\mathrm{M}}?x_g, \mathsf{rep}^{\mathrm{L}}?x_l)$ and let us repeat the experiments using the same parameters as above. The result is $\Xi_{\mathsf{install}}(\mathrm{H}) = 0$, $\Xi_{\mathsf{install}}(\mathrm{M}) = 0.28$, $\Xi_{\mathsf{install}}(\mathrm{L}) = 0.69$, and $\Xi_{\mathsf{install}}(\bot) = 0.03$. This shows that the risk of getting no schedule has not changed but that it is much more likely to be the local schedule than the global schedule.

5 Conclusion

Software is increasingly being used in malign rather than benign environments and this calls for adopting a more defensive programming style. Existing security errors are to a large extent due to an over-optimistic programming style where programmers focus on giving the software as much functionality as possible. The remedy is to adopt a more defensive programming style where programmers focus on avoiding errors that can be caused by external components — in particular, due to communication becoming unreliable.

There are numerous examples of software systems being developed in one context and then ported to another. In this process the threat scenario might change and changes to the software may be called for. As an example, the destruction of the Ariane 5 rocket on its maiden voyage was due to the reuse of software from Ariane 4 that suddenly operated outside of the previous design parameters, thereby causing a floating point overflow disrupting the safe control of the rocket.

We believe that suitable programming notations form a key ingredient in motivating programmers to produce more robust code. We also believe that such programming notations can be studied in a pure form, in the context of process calculi, as done in the present paper. In this paper the focus has solely been on the consequences of disruption to the communication links. This paves the way for dialects of main stream programming languages enforcing such robustness considerations and taking into account also the need to use cryptographic communication protocols to ensure confidentiality, integrity and authenticity of communication.

The main technical contribution of this paper is the development of a probabilistic trust analysis able to identify the extent to which the robust programming style has been adhered to.

The probabilistically based trust analysis indicates the places where decisions are made based on data of limited trustworthiness. It amounts to propagating the probability distributions from the binders throughout the program. This is done in such a way that the stochastic dependence between input variables is preserved and properly conditioned when passing through case statements. A naive implementation of probability distributions may lead to state explosion; more efficient methods may utilise symbolic data structures (like Binary Decision Diagrams) and sparse array techniques similar to those used in model checkers for Discrete Time Markov Chains.

In our view, this analysis forms the basis for supporting a new discipline of robust programming. We believe that it will be able to address the risks posed by porting software from one environment to another; by recalculating the probabilities one can better determine whether or not the software continues to deal appropriately with risks and threats in the new application environment.

Acknowledgement. The research has been supported by MT-LAB, a VKR Centre of Excellence for the Modelling of Information Technology, and by IDEA4CPS, funded by the Danish Foundation for Basic Research under grant DNRF86-10.

References

1. Bruni, R.: Calculi for service-oriented computing. In: Bernardo, M., Padovani, L., Zavattaro, G. (eds.) SFM 2009. LNCS, vol. 5569, pp. 1–41. Springer, Heidelberg (2009)
2. Guidi, C., Lucchi, R., Gorrieri, R., Busi, N., Zavattaro, G.: Sock: A calculus for service oriented computing. In: Dan, A., Lamersdorf, W. (eds.) ICSOC 2006. LNCS, vol. 4294, pp. 327–338. Springer, Heidelberg (2006)
3. Lapadula, A., Pugliese, R., Tiezzi, F.: A calculus for orchestration of web services. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 33–47. Springer, Heidelberg (2007)
4. Milner, R.: A Calculus of Communication Systems. LNCS, vol. 92. Springer, Heidelberg (1980)
5. Milner, R.: Communicating and Mobile Systems: the Pi-Calculus. Cambridge University Press (1999)
6. Nielson, F., Nielson, H.R., Hankin, C.: Principles of program analysis (2. corr. print). Springer (2005)
7. Nielson, H.R., Nielson, F., Vigo, R.: A Calculus for Quality. In: Păsăreanu, C.S., Salaün, G. (eds.) FACS 2012. LNCS, vol. 7684, pp. 188–204. Springer, Heidelberg (2013)
8. Wang, C., de Groot, M.: Managing end-user preferences in the smart grid. In: Proceedings of the 1st International Conference on Energy-Efficient Computing and Networking, e-Energy 2010, pp. 105–114. ACM (2010)