A Calculus for Quality

Hanne Riis Nielson, Flemming Nielson, and Roberto Vigo

DTU Informatics, Technical University of Denmark, Denmark
{riis,nielson,rvig}@imm.dtu.dk

Abstract. A main challenge of programming component-based software is to ensure that the components continue to behave in a reasonable manner even when communication becomes unreliable. We propose a process calculus, the Quality Calculus, for programming software components where it becomes natural to plan for default behaviour in case the ideal behaviour fails due to unreliable communication and thereby to increase the quality of service offered by the systems. The development is facilitated by a SAT-based robustness analysis to determine whether or not the code is vulnerable to unreliable communication. This is illustrated on the design of a fragment of a wireless sensor network.

Keywords: Distributed systems, availability of data, robustness, SAT-solving.

1 Introduction

One of the main challenges of component-based software development is to ensure that the distributed components continue to behave in a reasonable manner even when communication becomes unreliable. This is especially important for safety-critical software components in embedded systems and control software components that control part of our physical environment. With the advent of cyber-physical systems, in which software components are distributed throughout a physical system, the challenges will continue to grow in importance.

Considerable focus has been placed on how to ensure the integrity, confidentiality and authenticity of data communicated between components. In embedded systems this is easiest when communication takes place over cables shielded from other applications and used only for this purpose. However, increasingly cables are shared between many applications, including for example the infotainment system on cars, and often wireless communication needs to be employed as well, as when the control system needs to communicate with the pressure meter installed in the tyres. In health care applications there also is a trend to use wireless communication for interconnecting measuring apparatus with patient monitoring systems and with systems that dispense oxygen, saline or morphine. Solutions generally include the proper use of cryptographic communication protocols that can be proved secure using state-of-the-art analysis tools.

Less focus has been placed on how to ensure that the expected communication actually takes place. This is hardly surprising given the much more challenging

C.S. Păsăreanu and G. Salaün (Eds.): FACS 2012, LNCS 7684, pp. 188–204, 2013.

nature of this problem. One dimension of the problem is to ensure that other control components continue to operate and for this it often suffices to use model checking techniques for proving the absence of deadlock and livelock in software components. Another dimension is to ensure that messages sent are in fact received and this is much harder. Over the internet the possibility of denial of service attacks is well-known — simply flooding the internet with messages beyond the capacity of the recipient thereby masking the proper messages. Wireless communication is open to the same attacks as well as interference with the frequency band and physically shielding the antennas of sender and receiver as they are distributed throughout a cyber-physical system. Indeed, it might seem that this problem cannot be solved by merely using computer science techniques.

What is feasible using computer science techniques is to ensure that software systems are hardened against the unreliability of communication. This calls for programming software components of distributed systems in such a way that one has programmed a default behaviour to be enacted when the ideal behaviour is denied due to the absence of expected communication. To this end we propose

- a process calculus, the Quality Calculus, for programming software components and their interaction, and
- a SAT-based analysis to determine the vulnerability of the processes against unreliable communication.

The Quality Calculus is developed in Section 2 and clearly inherits from calculi such as CCS [9] and the π-calculus [10]. Its main novelty is a binder specifying the inputs to be performed before continuing. In the simplest case it is an input guard $t?x$ describing that some value should be received over the channel t and should be bound to the variable x. Increasing in complexity we may have binders of the form $\&_q(t_1?x_1, \cdots, t_n?x_n)$ indicating that several inputs are simultaneously active and 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 we distinguish between data and optional data, much like the use of option data types in programming languages like Standard ML. The construct $\mathsf{case}\ e\ \mathsf{of}\ \mathsf{some}(y) : P_1\ \mathsf{else}\ P_2$ will evaluate the expression e; 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 expressiveness of the Quality Calculus is considered in Section 3 and an example in the context of a wireless sensor network is presented in Section 4.

The SAT-based [8] robustness analysis is developed in Section 5. It is based on the view that processes must be coded in such a way that error configurations are not reached due to unreliable communication; rather, default data should be substituted for expected data in order to provide meaningful behaviour in all circumstances. Of course, this is not a panacea — default data is not as useful as the correct data, but often better quality of service may be obtained when basing decisions on default or old data, rather than simply stopping in an error state. As an example, if a braking system does not get information about the spinning of the wheels from the ABS system, it should not simply stop

Table 1. The syntax of the Quality Calculus

$$
\begin{array}{rcl}
P & ::= & (\nu c)\, P \;\mid\; P_1 \,|\, P_2 \;\mid\; \mathsf{0} \;\mid\; b.P \;\mid\; t_1!t_2.P \;\mid\; A(e) \\
& \mid & \mathsf{case}\ e\ \mathsf{of}\ \mathsf{some}(y) \colon P_1\ \mathsf{else}\ P_2 \\
b & ::= & t?x \;\mid\; \&_q(b_1, \cdots, b_n) \\
t & ::= & y \;\mid\; c \;\mid\; g(t_1, \cdots, t_n) \\
e & ::= & x \;\mid\; \mathsf{some}(t) \;\mid \mathsf{none} \;\mid\; f(e_1, \cdots, e_n) \\
& \mid & \mathsf{case}\ e\ \mathsf{of}\ \mathsf{some}(y) \colon e_1\ \mathsf{else}\ e_2
\end{array}
$$

braking, rather it should continue to brake — perhaps at reduced effect to avoid blocking the wheels. The analysis attaches propositional formulae to all points of interest in the processes; they characterise the combinations of optional data that could be missing. This is useful for showing that certain error configurations cannot be reached; indeed, if the propositional formula is unsatisfiable then the corresponding program point cannot be reached. The availability of extremely efficient SAT-solvers makes this a very precise analysis method with excellent scalability.

We conclude and present our outlook on future work in Section 6.

2 Syntax and Semantics

Process calculi are useful for delineating a programming abstraction that focuses on specific challenges in the development of distributed systems. Calculi such as CCS [9] and the π-calculus [10] have provided profound insights into the nature of concurrent computation.

There are a least two approaches to the use of process calculi. One focuses on the universality of calculi such as the π-calculus and would explain the computational paradigms of interest by their encoding into the π-calculus (which is known to be Turing complete). The other focuses on explaining the computational paradigms of interest as primitives in a suitable process calculus in order to avoid modelling artifacts, analysis artifacts, or other intricacies due to the encoding. The latter approach has lead to recent calculi such as COWS [7], SOCK [6], SCC [2] and CaSPiS [3] for understanding service-oriented computation and have suggested several novel paradigms for how to deal with services and the increasingly important notion of quality of service. We follow the latter approach in developing a process calculus, the Quality Calculus, that enforces robustness considerations on 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. A *system* consists of a number of process definitions and a main process:

$$\begin{array}{ll} \mathsf{define} & A_1(x_1) \triangleq P_1 \\ & \vdots \\ & A_n(x_n) \triangleq P_n \\ \mathsf{in} & P_* \end{array}$$

Here A_i is the name of a process, x_i is its formal parameter, P_i is its body and P_* is the main process. The syntax of processes is given in Table 1. A *process* can have the form $(\nu c)\, P$ 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 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 $A(e)$ to one of the defined processes with e being the actual parameter. Finally, a process can be a case construct whose explanation we defer to later. We shall feel free to dispense with trailing occurrences of the process $\mathsf{0}$.

The main novelty of the calculus is the *binder* b specifying the inputs to be performed before continuing. In the simplest case it is an input guard $t?x$ describing that some value should be received over the channel t and it will be bound to the variable x. Increasing in complexity we may have binders of the form $\&_q(t_1?x_1, \cdots, t_n?x_n)$ 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; these and other examples are summarised in Table 4. Even more complex cases arise when binders are nested, as in $\&_\forall(t_0?x_0, \&_\exists(t_1?x_1, t_2?x_2))$ that describes that input must be received over t_0 as well as one of t_1 or t_2. If we assume that our quality predicates can express all combinations of arguments then nested binders can always be unnested without changing the overall semantics; as an example $\&_\forall(t_0?x_0, \&_\exists(t_1?x_1, t_2?x_2))$ has the same effect as $\&_q(t_0?x_0, t_1?x_1, t_2?x_2)$ if $q(r_0, r_1, r_2)$ amounts to $r_0 \wedge (r_1 \vee r_2)$.

As a consequence, when continuing with the process P in $b.P$ some variables might not have obtained proper values as the corresponding inputs have not 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 terms t to denote data and expressions e 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 $\mathsf{case}\ e\ \mathsf{of}\ \mathsf{some}(y) : P_1\ \mathsf{else}\ P_2$ will test whether e evaluates to some data and if so, bind it to y and continue with P_1 and otherwise continue with P_2.

Clearly more elaborate choices of syntax for expressions and terms are possible including the possibility of distinguishing between them using type systems. However, for simplicity we have opted for two syntactic categories and therefore we also distinguish between functions g returning data values and functions f

Table 2. The structural congruence of the Quality Calculus

$P \equiv P$	$P_1 \equiv P_2 \Rightarrow P_2 \equiv P_1$	$P_1 \equiv P_2 \wedge P_2 \equiv P_3 \Rightarrow P_1 \equiv P_3$
$P \mid \mathsf{0} \equiv P$	$P_1 \mid P_2 \equiv P_2 \mid P_1$	$P_1 \mid (P_2 \mid P_3) \equiv (P_1 \mid P_2) \mid P_3$
$(\nu c)\, P \equiv P$ if $c \notin \mathsf{fc}(P)$	$(\nu c_1)\,(\nu c_2)\, P \equiv (\nu c_2)\,(\nu c_1)\, P$	$(\nu c)\,(P_1 \mid P_2) \equiv ((\nu c)\, P_1) \mid P_2$ if $c \notin \mathsf{fc}(P_2)$
$A(e) \equiv P[e/x]$ if $A(x) \triangleq P$	$P_1 \equiv P_2 \Rightarrow C[P_1] \equiv C[P_2]$	

returning optional data values. For expressions we additionally support a case construct much as for processes.

We need to impose a few well-formedness constraints on systems. For this we write $\mathsf{fc}(P)$ to denote the set of free constants in P, $\mathsf{fx}(P)$ to denote the set of free variables ranging over expressions, and $\mathsf{fy}(P)$ to denote the set of free variables ranging over terms. For a system of the form displayed above we require that $\mathsf{fx}(P_i) \subseteq \{x_i\}$, $\mathsf{fy}(P_i) = \emptyset$, $\mathsf{fx}(P_*) = \emptyset$, $\mathsf{fy}(P_*) = \emptyset$, and put no restrictions on $\mathsf{fc}(P_i)$ and $\mathsf{fc}(P_*)$.

Semantics. The semantics consists of a structural congruence and a transition relation [10]. 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. Finally, it allows replacement in contexts C given by:

$$C ::= [\,] \mid (\nu c)\, C \mid C \mid P \mid P \mid C$$

As usual, we apply α-conversion whenever needed in order to avoid accidental capture of names during substitution. 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 term t evaluates to a constant c and a similar relation describing when an expression e evaluates to a constant that either has the form $\mathsf{some}(c)$ or is none; the definitions of these relations are straightforward and hence omitted. Furthermore, we 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

$$b ::_v \theta$$

Table 3. The transition rules of the Quality Calculus

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

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

$$\frac{e \rhd \mathsf{some}(c)}{\mathsf{case}\ e\ \mathsf{of}\ \mathsf{some}(y) \colon P_1\ \mathsf{else}\ P_2 \longrightarrow P_1[c/y]}$$

$$\frac{e \rhd \mathsf{none}}{\mathsf{case}\ e\ \mathsf{of}\ \mathsf{some}(y) \colon P_1\ \mathsf{else}\ P_2 \longrightarrow P_2}$$

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

$$\frac{t_1 \rhd c_1}{c_1!c_2 \vdash t_1?x_2 \to [\mathsf{some}(c_2)/x_2]}$$

$$\frac{c_1!c_2 \vdash b_i \to b_i'}{c_1!c_2 \vdash \&_q(b_1, \cdots, b_i, \cdots, b_n) \to \&_q(b_1, \cdots, b_i', \cdots, b_n)}$$

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

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

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 substitution (θ) that has been constructed. To formalise this we extend the syntax of binders to include substitutions

$$b ::= \cdots \mid [\mathsf{some}(c)/x]$$

where $[\mathsf{some}(c)/x]$ is the substitution that maps x to $\mathsf{some}(c)$ and leaves all other variables unchanged. We 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. 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 group of clauses in Table 3 defines the auxiliary relation $c_1!c_2 \vdash b \to b'$. We have one clause for each of the two syntactic forms of b and the idea is simply to record the binding of the value received in the appropriate position.

Table 4. Quality predicates and their semantics

$$
\begin{array}{rl}
[\![\{\forall\}]\!](r_1,\cdots,r_n) = (|\{i \mid r_i = \mathsf{tt}\}| = n) & = r_1 \wedge \cdots \wedge r_n \\
[\![\{\exists\}]\!](r_1,\cdots,r_n) = (|\{i \mid r_i = \mathsf{tt}\}| \geq 1) & = r_1 \vee \cdots \vee r_n \\
[\![\{\exists!\}]\!](r_1,\cdots,r_n) = (|\{i \mid r_i = \mathsf{tt}\}| = 1) & \\
[\![\{m/n\}]\!](r_1,\cdots,r_n) = (|\{i \mid r_i = \mathsf{tt}\}| \geq m) &
\end{array}
$$

The auxiliary relation $b ::_v \theta$ is defined in the final group of clauses in Table 3. Here we perform a pass over the syntax of (the extended syntax of) the binder b evaluating whether or not a sufficient number of inputs has been performed (recorded in v) and computing the associated substitution θ. Table 4 gives examples of quality predicates to be used in the sequel together with their semantics; here we write $|X|$ for the cardinality of the set X.

Discussion. The semantics of Table 3 is a *rigid* semantics: The first time the top-level quality predicate holds the remaining inputs are no longer of interest and the computation can proceed. An alternative would be to use a *flexible* semantics and replace the two topmost rules of Table 3 with

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

The first clause expresses that we may continue accepting inputs even when $b ::_{\mathsf{tt}} \theta$ holds, that is, after the top-level quality condition is met the first time. The second clause ensures that at any point where the quality condition is met we can decide to proceed with the continuation process. Thus there is a non-deterministic choice as to how many inputs are accepted beyond the minimum number. This becomes a bit tricky when using quality predicates that do not satisfy a monotonicity requirement, meaning that the quality condition may go from true to false once more inputs have been accepted; this is for example the case for $\exists!$ in Table 4. On top of this important difference between the rigid and the flexible semantics, they also differ in their "speed"; as an example, in the rigid semantics a single step is needed to perform the binding of a single input whereas two steps are needed in the flexible semantics. Clearly the *flexible* semantics admits all the behaviours of the *rigid* semantics as well as sometimes additional ones.

3 Expressiveness of Binders

The binding operator $\&_q(b_1,\cdots,b_n)$ is surprisingly powerful and in this section we show how the primitives of the Quality Calculus can be used to define a number of other constructs known from process calculi. In the other direction the Quality Calculus can be encoded into the π-calculus but it would seem that some binding operators would require an exponential expansion; as an example, $\&_{n/2n}(b_1,\cdots,b_{2n})$ indicating that half of the $2n$ arguments are needed would seem to require that the π-calculus encoding would need to enumerate subsets of $\{1,\cdots,2n\}$ with at most n elements.

Guarded sum. Let us consider the guarded sum $\Sigma_{i=1}^{n} t_i?x_i.P_i$ of processes that each wants to perform an input before proceeding with their continuation. It can easily be encoded in our calculus using the binding construct:

$$\begin{aligned}\Sigma_{i=1}^{n} t_i?x_i.P_i \triangleq\ & \&_{\exists}(t_1?x_1, \cdots, t_n?x_n).\\ & (\mathsf{case}\ x_1\ \mathsf{of}\ \mathsf{some}(y_1)\colon P_1\ \mathsf{else}\ 0 \mid\\ & \qquad\vdots\\ & \mid \mathsf{case}\ x_n\ \mathsf{of}\ \mathsf{some}(y_n)\colon P_n\ \mathsf{else}\ 0)\end{aligned}$$

Here the quality predicate $\exists$ expresses that only 1 of the n inputs is required and we assume that no x_i occurs free in P_j when $i \neq j$.

To illustrate this in more detail let us consider the binary case $c_1?x_1.P_1 + c_2?x_2.P_2$ where the encoding amounts to:

$$\begin{aligned}c_1?x_1.P_1 + c_2?x_2.P_2 \triangleq\ & \&_{\exists}(c_1?x_1, c_2?x_2).\\ & (\mathsf{case}\ x_1\ \mathsf{of}\ \mathsf{some}(y_1)\colon P_1\ \mathsf{else}\ 0\\ & \mid \mathsf{case}\ x_2\ \mathsf{of}\ \mathsf{some}(y_2)\colon P_2\ \mathsf{else}\ 0)\end{aligned}$$

Let us assume that this process is in parallel with the process $c_1!c.Q$. Using Table 3 we have

$$c_1!c \vdash \&_{\exists}(c_1?x_1, c_2?x_2) \rightarrow \&_{\exists}([\mathsf{some}(c)/x_1], c_2?x_2)$$

and

$$\&_{\exists}([\mathsf{some}(c)/x_1], c_2?x_2) ::_{\mathsf{tt}} [\mathsf{some}(c)/x_1][\mathsf{none}/x_2]$$

so we get

$$c_1!c.Q \mid (c_1?x_1.P_1 + c_2?x_2.P_2) \longrightarrow Q \mid P_1[\mathsf{some}(c)/x_1][\mathsf{none}/x_2]$$

We have assumed that x_2 does not occur free in P_1 and hence we have the result we would expect.

Generalised input binder. We now introduce a version of the binding operator that even though it does not need all inputs in order to proceed still will honour them – and thereby ensure that other processes will not become stuck for that reason. The new binding operator is written $\&_q^?(t_1?x_1, \cdots, t_n?x_n)$ and is defined by

$$\begin{aligned}\&_q^?(t_1?x_1, \cdots, t_n?x_n).P \triangleq\ & \&_q(t_1?x_1, \cdots, t_n?x_n).\\ & (P \mid \mathsf{case}\ x_1\ \mathsf{of}\ \mathsf{some}(y_1)\colon 0\ \mathsf{else}\ t_1?x_1\\ & \qquad\vdots\\ & \mid \mathsf{case}\ x_n\ \mathsf{of}\ \mathsf{some}(y_n)\colon 0\ \mathsf{else}\ t_n?x_n)\end{aligned}$$

Thus the idea is to spawn processes in parallel to the continuation P taking care of the inputs that were not necessary according to the quality predicate.

To illustrate this let us consider the binary case $\&_{\exists}^?(c_1?x_1, c_2?x_2)$ where the encoding amounts to:

$$\begin{aligned}\&_{\exists}^?(c_1?x_1, c_2?x_2).P \triangleq\ & \&_{\exists}(c_1?x_1, c_2?x_2).\\ & (P \mid \mathsf{case}\ x_1\ \mathsf{of}\ \mathsf{some}(y_1)\colon 0\ \mathsf{else}\ c_1?x_1\\ & \mid \mathsf{case}\ x_2\ \mathsf{of}\ \mathsf{some}(y_2)\colon 0\ \mathsf{else}\ c_2?x_2)\end{aligned}$$

Assume that this process is in parallel with the process $c_1!c.Q_1$. Then we get

$$\begin{array}{l} c_1!c.Q_1 \mid \&^?_\exists(c_1?x_1, c_2?x_2).P \\ \quad \longrightarrow^* \; Q_1 \mid P[\mathsf{some}(c)/x_1][\mathsf{none}/x_2] \mid c_2?x_2 \end{array}$$

Thus the process $c_2?x_2$ is ready to take care of a late arrival of the input; so we will for example have

$$\begin{array}{l} c_2!c'.Q_2 \mid Q_1 \mid P[\mathsf{some}(c)/x_1][\mathsf{none}/x_2] \mid c_2?x_2 \\ \quad \longrightarrow Q_2 \mid Q_1 \mid P[\mathsf{some}(c)/x_1][\mathsf{none}/x_2] \end{array}$$

showing that the unsuccessful process $c_2!c'.Q_2$ will not be stuck even though its output is neglected.

Internal nondeterministic choice. We now show how to encode a version of the general sum $\bigoplus_{i=1}^n P_i$ of processes modelling *internal nondeterministic choice* between the alternatives. The idea is to introduce n fresh channels d_i over which a fresh constant d is communicated and bound to fresh variables x_i and y_i and then to select one of the summands:

$$\begin{array}{l} \bigoplus_{i=1}^n P_i \triangleq (\nu d_1) \cdots (\nu d_n)\,(\nu d) \\ \qquad (d_1!d \mid \cdots \mid d_n!d \\ \qquad \mid \&_\exists(d_1?x_1, \cdots, d_n?x_n). \\ \qquad\qquad (\mathsf{case}\ x_1\ \mathsf{of}\ \mathsf{some}(y_1)\colon P_1\ \mathsf{else}\ d_1?x_1 \mid \\ \qquad\qquad\quad \vdots \\ \qquad\qquad \mid \mathsf{case}\ x_n\ \mathsf{of}\ \mathsf{some}(y_n)\colon P_n\ \mathsf{else}\ d_n?x_n)) \end{array}$$

The difference from the ordinary CCS sum is that the choices are not made according to the availability of inputs but rather an internal nondeterministic choice is performed as in CSP.

Again let us consider the binary case where the encoding amounts to:

$$\begin{array}{l} P_1 \oplus P_2 \triangleq (\nu d_1)\,(\nu d_2)\,(\nu d) \\ \qquad (d_1!d \mid d_2!d \\ \qquad \mid \&_\exists(d_1?x_1, d_2?x_2). \\ \qquad\qquad (\mathsf{case}\ x_1\ \mathsf{of}\ \mathsf{some}(y_1)\colon P_1\ \mathsf{else}\ d_1?x_1 \\ \qquad\qquad \mid \mathsf{case}\ x_2\ \mathsf{of}\ \mathsf{some}(y_2)\colon P_2\ \mathsf{else}\ d_2?x_2)) \end{array}$$

Let us assume that it is $d_1!d$ that is successful and as above we get

$$d_1!d \vdash \&_\exists(d_1?x_1, d_2?x_2) \rightarrow \&_\exists([\mathsf{some}(d)/x_1], d_2?x_2)$$

and

$$\&_\exists([\mathsf{some}(d)/x_1], d_2?x_2) ::_{\mathsf{tt}} [\mathsf{some}(d)/x_1][\mathsf{none}/x_2]$$

and therefore we get

$$\begin{array}{l} P_1 \oplus P_2 \longrightarrow^* (\nu d_2)\,(\nu d)\,(d_2!d \mid P_1[\mathsf{some}(d)/x_1][\mathsf{none}/x_2] \mid d_2?x_2) \\ \quad \longrightarrow\ P_1 \end{array}$$

Here we have used that neither x_1, x_2, y_1 nor y_2 occur free in P_1 and that $d_2!d \mid d_2?x_2 \longrightarrow \mathsf{0}$.

Generalised output prefix. Finally we introduce an operator that allows a process to learn which outputs have been delivered and then use a quality predicate to determine when to proceed. The idea is to introduce new channels that can be used for internal communication when the outputs have been accepted. The new operator is denoted $\&^{!}_{q}(t_1!t'_1, \cdots, t_n!t'_n)$ and it is defined using the $\&^{?}_{q}(\cdots)$ binding operator introduced above:

$$\begin{aligned}\&^{!}_{q}(t_1!t'_1, \cdots, t_n!t'_n).P \triangleq\ & (\nu d_1) \cdots (\nu d_n)\,(\nu d) \\ & (t_1!t'_1.d_1!d \mid \cdots \mid t_n!t'_n.d_n!d \\ & \mid \&^{?}_{q}(d_1?x_1, \cdots, d_n?x_n).P)\end{aligned}$$

Here we assume that the new constants and variables do not occur in the terms t_i and t'_i nor in the process P. This operator will ensure that the continuation process P can start when some of the outputs have taken place (as determined by the quality predicate q) and it will also ensure that remaining outputs are still ready to be performed so that other processes do not get stuck because of missing communication possibilities.

To illustrate this let us consider the binary case $\&^{!}_{\exists}(c_1!c'_1, c_2!c'_2)$ where the encoding amounts to:

$$\begin{aligned}\&^{!}_{\exists}(c_1!c'_1, c_2!c'_2).P \triangleq\ & (\nu d_1)\,(\nu d_2)\,(\nu d) \\ & (c_1!c'_1.d_1!d \mid c_2!c'_2.d_2!d \\ & \mid \&_{\exists}(d_1?x_1, d_2?x_2). \\ & \quad (P \mid \mathsf{case}\ x_1\ \mathsf{of}\ \mathsf{some}(y_1){:}\ \mathsf{0}\ \mathsf{else}\ d_1?x_1 \\ & \quad\ \mid \mathsf{case}\ x_2\ \mathsf{of}\ \mathsf{some}(y_2){:}\ \mathsf{0}\ \mathsf{else}\ d_2?x_2))\end{aligned}$$

Assuming that this process is in parallel with the process $c_1?z_1.Q_1$ we get

$$\begin{aligned}& c_1?z_1.Q_1 \mid \&^{!}_{\exists}(c_1!c'_1, c_2!c'_2).P \\ & \quad \longrightarrow\ Q_1[c'_1/z_1] \mid (\nu d_1)\,(\nu d_2)\,(\nu d) \\ & \qquad\qquad (d_1!d \mid c_2!c'_2.d_2!d \\ & \qquad\qquad \mid \&_{\exists}(d_1?x_1, d_2?x_2). \\ & \qquad\qquad\quad (P \mid \mathsf{case}\ x_1\ \mathsf{of}\ \mathsf{some}(y_1){:}\ \mathsf{0}\ \mathsf{else}\ d_1?x_1 \\ & \qquad\qquad\quad\ \mid \mathsf{case}\ x_2\ \mathsf{of}\ \mathsf{some}(y_2){:}\ \mathsf{0}\ \mathsf{else}\ d_2?x_2)) \\ & \quad \longrightarrow^{*}\ Q_1[c'_1/z_1] \mid P \mid (\nu d_2)\,(\nu d)\,(c_2!c'_2.d_2!d \mid d_2?x_2)\end{aligned}$$

where we have used that neither x_1 nor x_2 occurs free in P. The resulting process is thus ready to handle the late communication over c_2; indeed we have

$$\begin{aligned}& c_2?z_2.Q_2 \mid Q_1[c'_1/z_1] \mid P \mid (\nu d_2)\,(\nu d)\,(c_2!c'_2.d_2!d \mid d_2?x_2) \\ & \quad \longrightarrow^{*}\ Q_2[c'_2/z_2] \mid Q_1[c'_1/z_1] \mid P\end{aligned}$$

showing that the additional machinery introduced ensures that all three processes can continue.

4 Motivating Example

We now consider a scenario inspired by [1] where a base station BS will communicate with a sensor node SN to obtain the value of a physical parameter,

which has to be communicated to a central aggregating unit. In order to ease the presentation, we will take the liberty to use a polyadic version of the calculus.

The sensor node SN is defined by

$$\begin{array}{l}\mathsf{SN} \triangleq \mathsf{0} \oplus (\mathsf{sn}?(x_i, x_p).\\ \quad \mathsf{case}\ x_i\ \mathsf{of}\ \mathsf{some}(y_i):\\ \quad\quad \mathsf{case}\ x_p\ \mathsf{of}\ \mathsf{some}(y_p): y_i!\mathsf{value}(y_p).\mathsf{SN}\ \mathsf{else}\ \mathsf{0}\\ \quad \mathsf{else}\ \mathsf{0})\end{array}$$

A basic node is equipped with a sensor able to measure one or more physical parameters (e.g. temperature, radioactivity) and a transceiver. As a node is typically powered by batteries, at some point in time it will die: this behaviour is captured by the possibility to non-deterministically evolve to 0 in the first line. While the node is alive, it waits for a request from the base station on channel sn, expecting the identity x_i of the sender and the name x_p of the parameter to be measured. The subsequent case constructs are used to extract the actual data, and then the measure is taken and communicated to the base station; the two else branches are in fact not reachable. The function value (which takes data as input and returns data) produces the result of measuring the intended parameter.

The base station will ask the sensor node to measure a physical parameter, and in the interest of its robustness we extend it with a process representing a local computer, able to estimate such a value. The local estimate will be communicated to the central unit and used whenever the sensor node does not respond. The local computer is defined by

$$\mathsf{LC} \triangleq \mathsf{lc}?x_e.\mathsf{case}\ x_e\ \mathsf{of}\ \mathsf{some}(y_e): \mathsf{lc}!\mathsf{guess}(y_e).\mathsf{LC}\ \mathsf{else}\ \mathsf{0}$$

and it uses the function guess (taking data and returning data) to estimate the value of the intended parameter; again, the case construct is used to extract the actual request and the else branch is not reachable.

The base station will put a limit on how long it will wait for a measure. In order to model this behaviour we make use of a time counter defined by

$$\mathsf{Clock} \triangleq \mathsf{set}?x_t.\mathsf{tick}!\checkmark.\mathsf{Clock}$$

where channel set is used to set a time-out, and the output of the constant ✓ signals that the prescribed amount of time has passed.

Finally, the base station is defined by the process

$$\begin{array}{l}\mathsf{BS} \triangleq (\nu id)\,(\nu p)\,(\nu t)\,\&^{!}_{\exists}(\mathsf{lc}!p, \mathsf{sn}!(id, p)).\mathsf{set}!t\\ \quad \&_{\forall}(\mathsf{tick}?x_t, \&^{?}_{\exists}(\mathsf{lc}?x_l, id?x_r))\\ \quad \mathsf{case}\ x_r\ \mathsf{of}\ \mathsf{some}(y_r): {}^{1}\mathsf{cu}!y_r.\mathsf{BS}\ \mathsf{else}\\ \quad\quad \mathsf{case}\ x_l\ \mathsf{of}\ \mathsf{some}(y_l): {}^{2}\mathsf{cu}!y_l.\mathsf{BS}\ \mathsf{else}\ {}^{3}\mathsf{0}\end{array}$$

where we have added some labels for later reference. In the first line the base station issues a request for a parameter p to the local computer and to the sensor node, identifying itself as id. The timer is set to the constant t as soon as one

of the recipients has received the request. The second line waits for the deadline and for at least one value among the local estimate and the real measure. This behaviour is determined by the top-most quality predicate $\forall$, which requires that both inputs are successful, and by the inner quality predicate $\exists$, which insists that at least one of its two inputs is successful. As we are using the binding operator $\&^{?}_{\exists}(\ldots)$ the other input will be handled when (and if) it arrives. It is important to note that it is also possible that both values arrive before the time has passed. The third line tests whether or not the sensor node responded; if this is the case the value is communicated to the central unit, otherwise the local estimate is sent. Observe that in this formalisation the final else branch (labelled 3) is not reachable, as the requests built by the base station correctly match the inputs of SN and LC, and the latter always responds.

Discussion. Let us conclude by discussing two alternative choices for the binding construct in the second line of BS. One possibility is to use the binder

$$\&_{2/3}(\mathsf{tick}?x_t, \mathsf{lc}?x_l, id?x_r)$$

and this would require that at least one entity among the sensor network and the local computer has communicated a value before proceeding. Another possibility is to use

$$\&_{\exists}(\mathtt{tick}?x_t, \&^{?}_{\exists}(\mathtt{lc}?x_l, id?x_r))$$

and in this case we might end up having no value at all.

5 Robustness Analysis

The Quality Calculus provides the means for expressing dure care in always having default data availble in case the real data cannot be obtained — but it does not enforce it.

Our enforcement mechanism will be a SAT-based [8] robustness analysis for characterising whether or not variables over optional data do indeed contain data. The analysis attaches propositional formulae to all points of interest in the processes; the formulae characterise the combinations of optional data that could be missing. At key places one would like to demand that such formulae would always require default data to be available; this translates into demanding that certain logical formulae are unsatisfiable as determined by a SAT-solver.

The formulae encode optional data as booleans as follows. A value of the form $\mathsf{some}(\cdot)$ is coded as tt and a value of the form none is coded as ff. We find it helpful to let $\overline{v}$ denote the boolean encoding of the value v, i.e. $\overline{\mathsf{some}(\cdot)} = \mathsf{tt}$ and $\overline{\mathsf{none}} = \mathsf{ff}$. As an example, the formula $x_1 \vee (x_2 \wedge x_3)$ indicates that either x_1 is available or both of x_2 and x_3 are available, the variables ranging over booleans.

The judgements. The main judgement of our analysis takes the form

$$\vdash \varphi \,@\, P$$

Table 5. Robustness Analysis of the Quality Calculus

$$\vdash \mathsf{tt} \mathbin{@} P_* \qquad \vdash \mathsf{tt} \mathbin{@} P_1 \qquad \cdots \qquad \vdash \mathsf{tt} \mathbin{@} P_n$$

$$\frac{\vdash \varphi \mathbin{@} (\nu c)\, P}{\vdash \varphi \mathbin{@} P} \qquad \frac{\vdash \varphi \mathbin{@} (P_1 \mid P_2)}{\vdash \varphi \mathbin{@} P_1} \qquad \frac{\vdash \varphi \mathbin{@} (P_1 \mid P_2)}{\vdash \varphi \mathbin{@} P_2}$$

$$\frac{\vdash \varphi \mathbin{@} (b.P) \quad \vdash b \blacktriangleright \varphi_b}{\vdash (\exists \mathsf{bv}(b).\varphi) \wedge \varphi_b \mathbin{@} P} \qquad \frac{\vdash \varphi \mathbin{@} (t_1 ! t_2.P)}{\vdash \varphi \mathbin{@} P}$$

$$\frac{\vdash \varphi \mathbin{@} (\mathsf{case}\ e\ \mathsf{of}\ \mathsf{some}(y) \colon P_1\ \mathsf{else}\ P_2) \quad \vdash e \triangleright \varphi_e}{\vdash \varphi \wedge \varphi_e \mathbin{@} P_1}$$

$$\frac{\vdash \varphi \mathbin{@} (\mathsf{case}\ e\ \mathsf{of}\ \mathsf{some}(y) \colon P_1\ \mathsf{else}\ P_2) \quad \vdash e \triangleright \varphi_e}{\vdash \varphi \wedge \neg\varphi_e \mathbin{@} P_2}$$

$$\frac{\vdash \varphi \mathbin{@} P}{\vdash (\exists x.\varphi) \mathbin{@} P}\ \text{if } x \in \mathsf{fv}(\varphi) \setminus \mathsf{fv}(P) \qquad \frac{\vdash \varphi \mathbin{@} P}{\vdash \varphi' \mathbin{@} P}\ \text{if } \varphi \Leftrightarrow \varphi'$$

$$\vdash t?x \blacktriangleright x \qquad \frac{\vdash b_1 \blacktriangleright \varphi_1 \quad \cdots \quad \vdash b_n \blacktriangleright \varphi_n}{\vdash \&_q(b_1, \cdots, b_n) \blacktriangleright [\![q]\!](\varphi_1, \cdots, \varphi_n)}$$

$$\vdash x \triangleright x \qquad \vdash \mathsf{some}(t) \triangleright \mathsf{tt} \qquad \vdash \mathsf{none} \triangleright \mathsf{ff}$$

$$\frac{\vdash e_1 \triangleright \varphi_1 \quad \cdots \quad \vdash e_n \triangleright \varphi_n}{\vdash f(e_1, \cdots, e_n) \triangleright [\![f]\!](\varphi_1, \cdots, \varphi_n)}$$

$$\frac{\vdash e_0 \triangleright \varphi_0 \quad \vdash e_1 \triangleright \varphi_1 \quad \vdash e_2 \triangleright \varphi_2}{\vdash \mathsf{case}\ e_0\ \mathsf{of}\ \mathsf{some}(y) \colon e_1\ \mathsf{else}\ e_2 \triangleright (\varphi_1 \wedge \varphi_0) \vee (\varphi_2 \wedge \neg\varphi_0)}$$

and the idea is that the formula φ describes the program point immediately before P. This is ambiguous in case there are multiple occurrences of the same subprocess in the system and the traditional solution is to add labels to disambiguate such occurrences but we dispense with this in order not to complicate the notation. The intended semantic interpretation of this judgement is that

$$\text{if} \vdash \varphi \mathbin{@} P \text{ and } P_* \rightarrow^* C[P\theta] \text{ then } \overline{\theta} \models \varphi$$

where $\overline{\theta}$ is the mapping obtained by pointwise application of the encoding $\overline{\cdot}$, and $\overline{\theta} \models \varphi$ denotes the truth of φ under the interpretation $\overline{\theta}$.

We will make use of two auxiliary judgements. One is for bindings

$$\vdash b \blacktriangleright \varphi$$

and the idea is that the formula φ describes the bindings of the variables that correspond to successful passing the binder b. The intended semantic interpretation of this judgement is that

$$\text{if } \vdash b \blacktriangleright \varphi \text{ and } b ::_{\mathsf{tt}} \theta \text{ then } \overline{\theta} \models \varphi$$

The other auxiliary judgement is for expressions; it takes the form

$$\vdash e \triangleright \varphi$$

and the idea is that the formula φ describes the result of evaluating the expression e. The intended semantic interpretation of this judgement is that

$$\text{if } \vdash e \triangleright \varphi \text{ and } e \rhd v \text{ then } \models (\varphi = \overline{v})$$

As usual, we write $\varphi_1 = \varphi_2$ as a shorthand for $(\varphi_1 \wedge \varphi_2) \vee (\neg\varphi_1 \wedge \neg\varphi_2)$.

The detailed definition. The formal definition of $\vdash \varphi \,@\, P$ is given by the inference system in the topmost part of Table 5. It operates in a top-down manner (as opposed to a more conventional bottom-up manner) and gets started by an axiom $\vdash \mathsf{tt} \,@\, P_*$ for the main process saying that it is reachable. Also we have an axiom for each of the defined processes; they have the form $\vdash \mathsf{tt} \,@\, P_i$ thereby ensuring that the process definitions are analysed in all contexts.

The first inference rule expresses that if φ describes the program point just before a process of the form $(\nu c)\, P$ then it also describes the program point just before P. Then we have two rules for parallel composition: if φ describes the program point before $P_1 \mid P_2$ then it also describes the program point just before each of the two processes. The rule for bindings in more interesting; here we make use of the auxiliary analysis judgement $\vdash b \blacktriangleright \varphi_b$ explained below for analysing the binding b. The information φ describing the program point before $b.P$ is transformed into $(\exists \mathsf{bv}(b).\varphi) \wedge \varphi_b$ in order to describe the program point before P; the existential quantification captures that potential free occurrences of the bound variables of b in φ are no longer in scope. The rule for output should now be straightforward. The two rules for the case construct make use of the auxiliary analysis judgement $\vdash e \triangleright \varphi_e$ explained below for analysing the expression; this gives rise to a formula describing the outcome of the test being performed and this information is added to describe the program point just before the selected branch.

Finally, we have two inference rules for manipulating the formulae describing the program points. The first one allows us to existentially quantify over variables not occurring free in the process being described. The second allows us to replace a formula with a logically equivalent one.

In the case of binders the formula φ produced by the judgement $\vdash b \blacktriangleright \varphi$ denotes that succesful passing of the binder gives rise to the formula φ for characterising the availability of data as provided by the binder. In the detailed definition of $\vdash b \blacktriangleright \varphi$ presented in the second part of Table 5 we rely on the formula schemes $[\![q]\!](r_1, \cdots, r_n)$ of Table 4 for encoding the effect of quality predicates q.

The last part of Table 5 defines the judgement $\vdash e \triangleright \varphi$ for expressions and as already mentioned the idea is that the formula φ characterises the availability of data used in e. Also here we rely on formula schemes of the form $[\![f]\!](r_1, \cdots, r_n)$ for encoding the effect of functions f and we assume that they satisfy the following soundness and completeness property:

$$[\![f]\!](\overline{v_1}, \cdots, \overline{v_n}) = \overline{v} \quad \text{whenever} \quad f(v_1, \cdots, v_n) \rhd v$$

Implementation. We have implemented this analysis by writing a program in Standard ML for computing the formulae at the program points of interest and

next use the SAT [8] and SMT [5] solver Z3 [4] to determine whether or not the formulae are satisfiable. For the examples we have studied the answer is obtained in less than a second on an ordinary laptop computer.

The motivating example. Let us return to the base station BS of Section 4 where we now want to compute the analysis results for the program points identified by the three labels. Starting with $\vdash \mathsf{tt} \,@\, \mathsf{BS}$ we obtain the following formulae at the labels:

$$\begin{array}{l} 1 : (x_1 \vee x_2) \wedge x_t \wedge (x_l \vee x_r) \wedge x_r \\ 2 : (x_1 \vee x_2) \wedge x_t \wedge (x_l \vee x_r) \wedge (\neg x_r) \wedge x_l \\ 3 : (x_1 \vee x_2) \wedge x_t \wedge (x_l \vee x_r) \wedge (\neg x_r) \wedge (\neg x_l) \end{array}$$

where we used the same variable names used in the process in order to stress the relationship between the formulae produced by the analysis and the program points they describe, even if here the variables range over the boolean encoding of optional data. Observe that $(x_1 \vee x_2)$ refer to the generalised output prefix $\&^{!}_{\exists}(\mathsf{lc}!p, \mathsf{sn}!(id, p))$ encoded as shown in Section 3, $(x_t \wedge (x_l \vee x_r))$ is the condition for passing the quality binder in the second line, and the remainder identifies the condition for reaching the given label. We can then ask whether or not the process points decorated with labels are reachable, that is, whether or not the related formulae are satisfiable. Using Z3 we obtain the following satisfying substitutions:

$$\begin{array}{l} 1 : [x_2 \mapsto \mathsf{ff}; x_1 \mapsto \mathsf{tt}; x_l \mapsto \mathsf{tt}; x_r \mapsto \mathsf{tt}; x_t \mapsto \mathsf{tt}] \\ 2 : [x_2 \mapsto \mathsf{ff}; x_1 \mapsto \mathsf{tt}; x_l \mapsto \mathsf{tt}; x_r \mapsto \mathsf{ff}; x_t \mapsto \mathsf{tt}] \\ 3 : \text{not satisfiable} \end{array}$$

This shows that the $\mathsf{0}$ process of BS will never be executed.

Let us conclude by considering the variants of the base station discussed at the end of Section 4. Using the binder $\&^{?}_{2/3}(\mathsf{tick}?x_t, \mathsf{lc}?x_l, id?x_r)$ we get slightly different formulae but the satisfiability results are the same as above: the formula for the process labelled 3 is unsatisfiable whereas the others have satisfying assignments.

Using the binder $\&_{\exists}(\mathsf{tick}?x_t, \&^{?}_{\exists}(\mathsf{lc}?x_l, id?x_r))$ we get the following formula for the process labelled 3:

$$3 : (x_1 \vee x_2) \wedge (x_t \vee x_l \vee x_r) \wedge (\neg x_r) \wedge (\neg x_l)$$

which is satisfiable using the substitution:

$$3 : [x_2 \mapsto \mathsf{ff}; x_1 \mapsto \mathsf{tt}; x_l \mapsto \mathsf{ff}; x_r \mapsto \mathsf{ff}; x_t \mapsto \mathsf{tt}]$$

The $\mathsf{0}$ process labelled 3 might thus be reachable. The above substitution gives us an indication of when this can happen: the binder $\&_{\exists}(\mathsf{tick}?x_t, \&^{?}_{\exists}(\mathsf{lc}?x_l, id?x_r))$ will be successful when $x_t = \mathsf{tt}$ meaning that the time has passed but it does not need to be the case that any of the schedules are available as reflected by $x_l = \mathsf{ff}$ and $x_g = \mathsf{ff}$. In this case the $\mathsf{0}$ process will in fact be reached and the BS process will terminate.

Formal correctness. We have argued informally that the analysis is correct with respect to the semantics and this is in line with how static analyses of programming languages are often presented. The main obstacle in giving a formal proof of correctness is that the semantics applies substitutions directly whereas the correctness statements talk about explicit substitutions. This is a well-known obstacle and at least two solutions are possible. One is to keep the semantics and correctness statements and to emulate the technically complex approach of [11]. Another is to modify the semantics to use explicit substitutions and perform a more direct proof of correctness leaving the technical complexities to proving the equivalence of the original semantics to the modified semantics. However, this technically complex development would provide little additional insight onto our approach.

6 Conclusion

Many of the errors in current software are due to an overly optimistic programming style. Programmers tend to think of benign application environments and hence focus on getting the software to perform as many functions as possible. To a much lesser extent they consider malign application environments and the need to focus on avoiding errors that can be provoked by outside attackers.

This is confounded by the fact that key software components are often developed in one context and then ported to another. The Simple Mail Transfer Protocol (SMTP) is a case in point. Originally developed in benign research or development environments, where few would be motivated to misuse the protocol and could easily be reprimanded if doing so, it has become a key constituent of the malign environment provided by the global internet where many users find an interest in misusing the protocol, and where it is extremely difficult to even identify offenders.

Future programming languages and programming environments need to support a more robust (pessimistic) programming style: What conceivably might go wrong probably will go wrong. A major cause of disruption is due to the communication between distributed sofware components. There is an abundant literature on methods and techniques for how to prevent attackers from learning secrets (confidentiality) or from telling lies (integrity, authenticity). Hence our focus considers how to mitigate the consequences of attackers, nature or misfortune preventing expected communication from taking place. This calls for a very robust way of programming systems where there always are default data available for allowing the system to continue its operation as best as it can (rather than simply terminate with an error or get stuck in an input operation).

We believe that the Quality Calculus presents the core ingredients of a process calculus supporting such defensive (robust) programming. To assist in analysing the extent to which robustness has been achieved we have developed a SAT-based robustness analysis, that indicates the places where errors can still arise in spite of robust programming, and where additional hardening of the code may be called for.

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

References

1. Anand, M., Ives, Z., Lee, I.: Quantifying eavesdropping vulnerability in sensor networks. In: Proceedings of the 2nd International Workshop on Data Management for Sensor Networks, DMSN 2005, pp. 3–9. ACM (2005)
2. Boreale, M., Bruni, R., Caires, L., De Nicola, R., Lanese, I., Loreti, M., Martins, F., Montanari, U., Ravara, A., Sangiorgi, D., Vasconcelos, V., Zavattaro, G.: SCC: A Service Centered Calculus. In: Bravetti, M., Núñez, M., Zavattaro, G. (eds.) WS-FM 2006. LNCS, vol. 4184, pp. 38–57. Springer, Heidelberg (2006)
3. 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)
4. de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
5. de Moura, L., Bjørner, N.: Satisfiability modulo theories: introduction and applications. Commun. ACM 54(9), 69–77 (2011)
6. 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)
7. 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)
8. Malik, S., Zhang, L.: Boolean satisfiability from theoretical hardness to practical success. Commun. ACM 52(8), 76–82 (2009)
9. Milner, R.: A Calculus of Communication Systems. LNCS, vol. 92. Springer, Heidelberg (1980)
10. Milner, R.: Communicating and Mobile Systems: the Pi-Calculus. Cambridge University Press (1999)
11. Nielson, F., Nielson, H.R., Bauer, J., Nielsen, C.R., Pilegaard, H.: Relational Analysis for Delivery of Services. In: Barthe, G., Fournet, C. (eds.) TGC 2007. LNCS, vol. 4912, pp. 73–89. Springer, Heidelberg (2008)