The Probabilistic Steam Boiler: a Case Study in Probabilistic Data Refinement

Annabelle McIver, Carroll Morgan[⋆] and Elena Troubitsyna[⋆⋆]

Programming Research Group, Oxford University
and Turku Centre for Computer Science (TUCS), Åbo Akademi.

Abstract. Probabilistic choice and demonic nondeterminism have been combined in a model for sequential programs [11, 8] in which program refinement is defined by removing demonic nondeterminism. Here we study the more general topic of *data* refinement in the probabilistic setting, extending standard techniques to probabilistic programs. We use the method to obtain a quantitative assessment of the safety of a (probabilistic) version of the steam boiler [1].

Keywords: Probabilistic semantics, verification, refinement, data refinement, imperative programming, safety assessment, system safety.

1 Introduction

One datatype is said to be refined by another if the second can replace the first in any program without detection. Techniques for proving refinements are well established [10, 4] and involve setting up a correspondence between the (abstract) states in the specification datatype and the (concrete) states in the implementation datatype. The standard techniques however are inadequate when the datatypes (and the programs that use them) contain probabilistic choice — for that we need a correspondence between *probability distributions* over abstract states and *probability distributions* over concrete states, rather than between individual state-to-state relations.

In this paper we show how the standard methods can be extended in that way, to allow for probabilistic data refinements. Probability is of particular relevance when a quantified analysis of safety is required — thus we treat the 'steam boiler' as our illustrative example, which is widely known [1] as a test case for formal system specification.

In Sec. 2 we set out the standard definitions for program refinement and data refinement in the probabilistic context. Sec. 3 contains our first

⋆ McIver and Morgan are members of the Programming Research Group: {*anabel,carroll*}*@comlab.ox.ac.uk*; McIver is supported by the EPSRC.

⋆⋆ Troubitsyna is a graduate student at the Turku Centre for Computer Science (TUCS): *etroubit@ra.abo.fi*.

main technical contribution — a completely worked example of a probabilistic datatype consisting of a single operation which chooses probabilistically either to skip or to abort. It is of the kind lying strictly outside the domain of standard programming, and its behavioural simplicity belies the subtleties of the calculations required to prove data refinement. That then sets the scene for Sec. 4 in which we look at the steam boiler proper, and though much simplified it serves our purpose well, which is to demonstrate how probabilistic datatypes (such as the one studied in Sec. 3) are relevant in a realistic setting. Throughout we assume only minimal knowledge of probabilistic program semantics, and where necessary the details are carefully explained with full definitions cited in other sources.

In general, such refinements can be viewed in two ways. One way is to accept the probabilistic behaviour of the components and then to determine by calculation what is the best specification that can be achieved of a system built from them; the other way is to deduce from a specification the minimum acceptable behaviour of the components from which it is to be built. In this study we treat refinement as an exercise of the former.

2 Data Refinement and Simulations

Data refinement is a generalised form of program refinement — an 'abstract datatype' is replaced by a more 'concrete datatype' in a program while preserving its algorithmic structure, the difference being that the two datatypes operate over different (local) state spaces and thus cannot be related directly using ordinary program refinement. In this context we refer to a *datatype* (as do Gardiner and Morgan [5]) as a triple (I, OP, F) where I and F are programs (respectively the initialisation and the finalisation) and OP is an indexed set of programs — the operations of the datatype.

Typically I initialises the local variables, possibly but not necessarily referring to global variables in doing so; operations in OP may refer to and change both local and global variables; and F simply 'discards' the final values of the local variables, projecting the state (local,global) back onto its purely global component.

In the activity of data refinement the programs I, F and those in OP are replaced by corresponding concrete programs, perhaps changing the local state space in the process. The effect of replacement on a program using the datatype is proper program refinement between that using the abstract type and that using the concrete, provided the local state cannot be observed directly — an assumption that is vital for the discussion of data refinement.

In this paper we consider datatypes whose operations may have probabilistic choices, and thus our model for programs is a relational-style structure augmented with probabilities [11, 8]. As in standard relational models, a program is represented by a relation between (initial) states and possibly many (final) states. But in our model we include the probabilistic structure underlying our programs' behaviours by taking the 'final states' in fact to be probability distributions over the underlying state space. Denoting that underlying space by S we define $\overline{S}$ to be the set of *discrete probability distributions* over S, thus

$$\overline{S} \quad := \quad \{F\colon S \to [0,1] \mid \sum_{s:S} F.s \leq 1\} \ .$$

Probabilistic programs now form a subset of the space of functions $S \to \mathbb{P}\overline{S}$.[1] The multiplicity of the result set means that programs generalise Markov processes (which are of type $S \to \overline{S}$ from elementary probability theory [6]), and as such produce *ranges* of probability distributions — that, as we shall see, provides a convenient way to specify 'safety boundaries'.

Program refinement ($\sqsubseteq$) works as expected by reducing those ranges: if for programs P, Q we have $P \sqsubseteq Q$, then any distribution (over final states) that Q can produce must also potentially be a result distribution of P — also, the probability that Q terminates (does not abort) must be least as great as the probability that P does.

We define data refinement with respect to program refinement.

Definition 2.1. We say (I, OP, F) is *data refined* by (I', OP', F') if and only if for all program schemes $\mathcal{P}$ over the operations OP of the datatype we have

$$I; \mathcal{P}(OP); F \quad \sqsubseteq \quad I'; \mathcal{P}(OP'); F' \ ,$$

where ';' denotes sequential composition . □

Note that the effect of prepending and appending respectively the initialisation and the finalisation make a composite that refers only to global variables — the local variables are hidden in the sequential compositions of Def. 2.1. Hence the refinement relation shown is ordinary program refinement between programs over the globally observed state space.

[1] For a full motivation, definition and discussion of the probabilistic model for sequential programs refer to Morgan *et al.* [11].

The scheme $\mathcal{P}$ is formed from the programming language constructs (and takes arguments from the appropriate datatypes) — we use the language of guarded commands [3] extended [11, 8] by a binary probabilistic choice operator ${}_p\oplus$, where $p \in [0,1]$.[2] Thus for example the probabilistic assignment

$$i\colon= i+1 \quad {}_p\oplus \quad i\colon= 0 \ , \tag{1}$$

means that the standard assignments $i\colon= i+1$ and $i\colon= 0$ are selected with probability p and $1{-}p$ respectively. The effect of the statement (1) thus depends on the initial value of the variable i, and for each one there is only a single result distribution — namely that which is weighted p or $1{-}p$ between the (final) values i_0+1 or 0 respectively (where i_0 denotes the initial state of i).

We introduce other program constructs as we need them.

Rather than using the cumbersome Def. 2.1 directly — for which we would need to check refinement *for all* functions $\mathcal{P}$ — to verify data refinements we use instead the standard technique of simulations [9].

Definition 2.2. We say that (I', OP', F') *simulates* (I, OP, F) if there is a program $rep\colon S \to \mathbb{P}\overline{S}$ satisfying:

$$\begin{array}{lcl} I; rep & \sqsubseteq & I' \\ opa; rep & \sqsubseteq & rep; opc \\ F & \sqsubseteq & rep; F' \ , \end{array}$$

where the second refinement holds for all corresponding pairs $(opa,\ opc)$ in $OP \times OP'$. □

It is well known [4] that a simulation of datatypes implies data refinement Def. 2.1; the same holds in the probabilistic model, provided *rep* is suitably defined.[3]

In the next section we use the simulation technique in a worked example.

[2] In general p can be a function of the state; but in this presentation we will need only constants.

[3] For *rep* to be a simulation that implies data refinement between datatypes it must satisfy two conditions. The first is continuity (as a function over the program domain); and the second is that it must not affect the global state space. The proof then of the soundness of the simulation technique follows exactly that explained by Gardiner [4] for standard programs, save for distribution of *rep* through external demonic nondeterminism. However distribution through even that operator can be established provided the locality of variables is modelled explicitly [7].

3 Probabilistic Datatypes: a Worked Example

The example we consider in this section is set out in Fig. 1. The abstract — or specification — datatype has a single operation which either skips with probability a or aborts with probability $1-a$ (written $\overline{a}$). The concrete (implementation) datatype has a local variable i; the operation opc_N first checks whether the variable i exceeds $N-1$, aborting if it does so and skipping otherwise, which behaviour is achieved by the assertion statement $\{i < N\}$. After that it either increments i (probability c_N) or sets it to 0 (probability $\overline{c_N}$). Thus in the concrete datatype abortion is possible only after the left branch has been selected N times in succession.

$$\begin{array}{rl} \mathit{inita} & \textbf{skip} \\ \mathit{opa} & \textbf{skip} \;{}_a\!\oplus\; \textbf{abort} \\ \mathit{fina} & \textbf{skip} \end{array}$$

$$\begin{array}{rl} \mathit{initc} & i{:=}\,0 \\ \mathit{opc}_N & \{i < N\};\;\; (i{:=}\, i+1 \;{}_{c_N}\!\oplus\; i{:=}\,0) \\ \mathit{finc} & \textbf{skip} \end{array}$$

Fig. 1. A faulty N-**skip**per

The qualitative external behaviour of the two datatypes is identical — each provides a single operation which acts either like **skip** or like **abort**. Our task however is quantitative — to determine the relation between a and c_N which ensures that opc_N is no more likely to abort than opa is.

The object of this section is carefully to work through the calculation of probabilities required to establish a simulation between the datatypes, and since our aim is merely to illustrate the technique we treat the simplest interesting case — when $N = 2$ — as is set out in Fig. 2. We return to the general case at the end of the section.

For the simple 2-**skip**per, according to Def. 2.2, we need a program *rep* satisfying the refinements[4]

$$\begin{array}{rcl} \mathit{inita}; \mathit{rep} & \sqsubseteq & \mathit{initc} \\ \mathit{opa}; \mathit{rep} & \sqsubseteq & \mathit{rep}; \mathit{opc}_N \\ \mathit{fina} & \sqsubseteq & \mathit{rep}; \mathit{finc} \end{array} \qquad (2)$$

[4] In fact we shall concentrate only on the first two, for the refinement $\mathit{fina} \sqsubseteq \mathit{rep}; \mathit{finc}$ is trivially satisfied provided the program *rep* terminates — and ours does.

The program *rep* can be thought of as calculating concrete states for the corresponding abstract states.

$$\begin{array}{rl}
\mathit{inita} & \mathbf{skip} \\
\mathit{opa} & \mathbf{skip}\ {}_a\!\oplus\ \mathbf{abort} \\
\mathit{fina} & \mathbf{skip} \\
\\
\mathit{initc} & i{:}=0 \\
\mathit{opc} & \{i<2\};\ (i{:}=i+1\ {}_c\!\oplus\ i{:}=0) \\
\mathit{finc} & \mathbf{skip}
\end{array}$$

Fig. 2. A faulty 2-**skip**per

We note first that a successful execution of *opa* must correspond to a successful execution of *opc*, which means i initially takes the values 0 or 1 — the initial assertion $\{i < 2\}$ in *opc* ensures that — and thus $0, 1$ or 2 finally in some distribution. Bearing in mind that *rep* 'converts' abstract states into concrete states, we might imagine that it sets i to some distribution of those outcomes. Thus we suppose that *rep* has the form[5]

$$\left|\begin{array}{l} i{:}=0\ @\ p \\ i{:}=1\ @\ q \\ i{:}=2\ @\ r \end{array}\right. ,$$

for some 1-summing p, q, r to be determined. Next we substitute into the simulation equations (2) above to get

$$\begin{array}{l}
\mathbf{skip}\ {}_a\!\oplus\ \mathbf{abort}; \\
\left|\begin{array}{l} i{:}=0\ @\ p \\ i{:}=1\ @\ q \\ i{:}=2\ @\ r \end{array}\right.
\end{array}
\quad\sqsubseteq\quad
\begin{array}{l}
\left|\begin{array}{l} i{:}=0\ @\ p \\ i{:}=1\ @\ q \\ i{:}=2\ @\ r; \end{array}\right. \\
\{i<2\};\ i{:}=i+1\ \ {}_c\!\oplus\ \ i{:}=0
\end{array}
\qquad (3)$$

Then we simplify by computing the various sequential compositions along the probabilistic branches, multiplying probabilities as we go (a list of algebraic laws we use is presented in the appendix); on the left the result

[5] For convenience we write such extended probabilistic choices with the components' individual probabilities given explicitly on the right, rather than as a nesting of binary ${}_p\!\oplus$ operators.

is

$$\left|\begin{array}{ll} \textbf{skip};\quad i{:=}\,0 & @\ ap \\ \textbf{abort};\, i{:=}\,0 & @\ \overline{a}p \\ \textbf{skip};\quad i{:=}\,1 & @\ aq \\ \textbf{abort};\, i{:=}\,1 & @\ \overline{a}q \\ \textbf{skip};\quad i{:=}\,2 & @\ ar \\ \textbf{abort};\, i{:=}\,2 & @\ \overline{a}r \end{array}\right. .$$

Simplifying the program fragments, and (as a result) collapsing the three aborting cases, gives

$$\left|\begin{array}{l} i{:=}\,0\ @\ ap \\ i{:=}\,1\ @\ aq \\ i{:=}\,2\ @\ ar \end{array}\right. ,$$

where for further brevity we have adopted the convention that when the probabilities sum to less than 1 the deficit represents aborting behaviour. Similar calculations on the right show us that (3) is equivalent to

$$\left|\begin{array}{l} i{:=}\,0\ @\ ap \\ i{:=}\,1\ @\ aq \\ i{:=}\,2\ @\ ar \end{array}\right. \quad\sqsubseteq\quad \left|\begin{array}{l} i{:=}\,0\ @\ \overline{c}(p+q) \\ i{:=}\,1\ @\ cp \\ i{:=}\,2\ @\ cq \end{array}\right.$$

from which we must extract a relation between a, p, q, r and c.

The refinement relation $\sqsubseteq$ applied between probabilistic programs requires that any outcome guaranteed on the left with a certain probability is guaranteed on the right with a probability at least as great. In this case we have that when i assigned a particular value on the left-hand side the associated probability must be at least as great as for the same assignment on the right-hand side. So matching the cases $i = 0, 1, 2$ separately we end up with the inequations

$$\begin{array}{l} ap \le \overline{c}(p+q) \\ aq \le cp \\ ar \le cq \ , \end{array}$$

which are the constraints we seek.

As mentioned in the introduction, there are now two approaches we could take. In the 'client-oriented' approach the specification is fixed — the client demands a certain reliability — and the supplier has to find components that are themselves reliable enough to meet it; in our current example we would fix a and calculate how high c would then have to be.

In the 'supplier-oriented' approach we imagine an implementor working with 'off the shelf' components and calculating the impact from the client's point of view. That is the view we take here: thus we fix c and find the largest a for which the equations can be satisfied by some p, q, r. With some algebra we see that the optimal value for a occurs when

$$a \quad = \quad \overline{c} + \sqrt{1 + 2c - 3c^2}/2 \ ,$$

attained at $p = \overline{c}/a$, $q = cp/a$ and $r = 0$. When c for example is $1/2$, the corresponding a for the above is $(\sqrt{5}+1)/4$, obtained by giving p the value $(\sqrt{5}-1)/2$; thus we are using a *rep* defined as

$$i\!:= 0 \quad {}_{(\sqrt{5}-1)/2}\!\oplus \quad i\!:= 1$$

to show the refinement of Fig. 3.

inita	**skip**		
opa	**skip** ${}_{(\sqrt{5}+1)/4}\!\oplus$ **abort**	$\leftarrow$	$(\sqrt{5}+1)/4 \approx .81$
fina	**skip**		
initc	$i\!:= 0$		
opc	$\{i < 2\};\ i\!:= i + 1 \ {}_{0.5}\!\oplus i\!:= 0$		
finc	**skip**		

Fig. 3. A faulty **skip**per

Finally, to deal with the initialisation we use the standard technique of composing data refinements [2]: reducing the value of i will make abortion less likely in the concrete datatype, and that is achieved with a second representation function $i\!:\leq i$ — that is, a (demonic) nondeterministic assignment that cannot increase the value of i. Thus the final form for *rep* is given by a sequential composition

$$rep \quad := \quad (i\!:= 0 \ {}_{p}\!\oplus i\!:= 1);\ i\!:\leq i\ ;$$

and from that *rep* it can be shown that *all* refinements in (2) hold.

But why is the abstract a not equal to $1-(.5)^2 = .75$, simply the probability that the concrete datatype does not perform two increments of i successively? Instead its calculated value of .81 shows the module to be slightly more reliable than we had expected.

In general, the naive view would expect a to be $\overline{c^2}$. But by tabulating the incremental probability of success (non-abortion) for both datatypes, Fig. 4 shows that intuition is indeed pessimistic in this case: after two calls to *opa* (*opc*), for example, we should compare $\overline{c^2}$ — the chance that $i < 2$ — with a^2, rather than with a. That illustrates the benefits of exact calculation, aside from a proof of correctness.

	s_a	s_c	
inita; *opa*	a	1	*initc*; *opc*
inita; *opa*; *opa*	a^2	$\overline{c^2}$	*initc*; *opc*; *opc*
inita; *opa*; *opa*; *opa*	a^3	$\overline{c}(1 + c\overline{c})$	*initc*; *opc*; *opc*; *opc*
⋮	⋮	⋮	⋮

The columns s_a, s_c give the cumulative success probability for the abstract, concrete datatype — the probability that abort does not occur in the given sequence of operations. The resulting constraint on a is that in all rows the value in the left column must not exceed the corresponding value in the right column.

Fig. 4. Tabulating the chance of success

Finally we generalise the above ideas to treat the N-**skip**per of Fig. 1. Following the same reasoning as above we use

$$rep \quad := \quad (|n : 0 \le n \le N : i := n \ @ \ p_n); \ \ i :\le i \ , \tag{4}$$

where $p_0, p_1, \cdots, p_N$ are probabilities summing to 1. As before, when the refinement equations are simplified we recover a set of constraints:

$$\begin{array}{ll} ap_0 & \le \overline{c_N} \\ ap_{n+1} & \le c_N p_n \text{ for } 0 \le n < N \ . \end{array} \tag{5}$$

Notice that this time the assignment to i is chosen from $0, ..., N$ since if opc_N does not abort, i must have one of these values finally. Once more we maximise for a, the result of which is summarised in the following lemma.

Lemma 3.1. The program *rep* defined in (4) is a simulation between the datatypes in Fig. 1 provided $a \le a'$, where a' is the largest solution for x in the interval $[0, 1]$ of the equation

$$x^N \quad = \quad \overline{c}(x^{N-1} + cx^{N-2} + ... + c^{N-2}x + c^{N-1}) \ . \tag{6}$$

Proof: A simulation holds between the datatypes provided the equations (5) are satisfied and maximising for a implies finding solutions of the equation (6). It now follows that, since (5) hold for equality for a', they also hold (for inequality) for any smaller value of a. □

In the general case there is always a unique solution of the equation (6) lying in the interval $[0, 1]$. (We are interested in that interval since we look for probabilities.) A careful analysis shows that it always allows us to prove a data refinement with a somewhat larger than $1 - (c_N)^N$, the probability that i is incremented N times in a row.

4 An Application: a Probabilistic Steam Boiler

In this section we consider the problem of calculating the reliability of a regulator for a steam boiler. An accurate model would be much more complicated than the one presented here — but we concentrate on only a small number of details, many of which we simplify later, for our primary aim is to illustrate how the data refinement techniques developed in the previous sections might apply to a realistic safety-analysis problem.

4.1 System Safety

The steam boiler consists of a reservoir of water supplied by pumps. During operation the water level continually fluctuates in reaction to steam loss and varying pump and water pressure. If the water level becomes either too low or too high the boiler could be seriously damaged, thus we describe our system reliability in terms of the level's remaining between a low (L) and a high (H) boundary. We use the standard safety engineering definition of *unreliability* [14] — it is the proportion of observed failures in a given interval of time (*reliability* is the complement). Thus in this application our unreliability measures appear as a probability of failure per unit time; each unit is a single iteration of one of the programs given below.

The steam boiler has a *regulator*, which we write as a datatype[6] specifying the overall tolerance on system reliability, and which otherwise has very simple behaviour — it abstracts from the sensors and the fault tolerance mechanism which would appear in the more complicated implementation. By proving refinement between specification and implementation we aim to establish a link (as in standard safety analysis) between high-level 'system reliability' and low-level component failure rate. We concentrate on the two system components that together contribute to unreliability by independently affecting the water level: they are the *environment* and the regulator, and we describe them next.

[6] In fact we present only the operations of the datatypes — it is a routine exercise to include the initialisation and finalisation.

4.2 The Environment

Our intended environment is a very general one, for it can influence the water level in either direction. Disturbances in the ambient temperature, for example, impact on the rate of steam production and can cancel the effect of pump action. In Fig. 5 we illustrate the possible changes in water level during a single unit of time.

steam	**pump**	**simplified effect δ**
not outgoing	off	0
not outgoing	on	+1
outgoing	on	0
outgoing	off	−1

Fig. 5. Change in water level during a single unit of time

In a more elaborate system model we might abstract specific environment reactions by a nondeterministic choice over a range restricted only by the physical constraints of pump and steam pressure. Here our aim is more modest, and we simplify matters by assuming that the rate of water movement δ is either constant or zero; normalisation then makes it -1, 0 or $+1$. With that abstraction the environment's behaviour is very simple: it is the nondeterministic choice ($\sqcap$) over the possibilities of water-level movement δ in one time unit:

$$\delta := -1 \;\sqcap\; \delta := 0 \;\sqcap\; \delta := 1 \ .$$

4.3 Safety Specification for the Regulator

The behaviour of the steam boiler system is to alternate between the environment action and the regulator reaction. A single step of such a cycle is therefore given by

$$\begin{array}{l} \delta := -1 \sqcap \delta := 0 \sqcap \delta := 1; \\ \mathit{Regulator}(w, \delta); \\ \{L \leq w \leq H\} \ . \end{array}$$

The task of the procedure $\mathit{Regulator}(w, \delta)$ is to keep the water level w between the boundaries H and L, possibly counteracting the environment (ignoring δ) to do so — because if it fails, the assertion statement $\{L \leq w \leq H\}$ will cause the whole system to abort.

The specification of the regulator set out in Fig. 6 describes both correct and faulty behaviour, linked by probabilistic choice, and thus defines the overall tolerance placed on the regulator's (and hence the system's) reliability. We have used statements $P\ {}_{\geq p}\!\oplus\ Q$ which combine probability and nondeterminism — P is chosen with some probability *at least* p, but we don't know which.[7] For our use of it here, all we need to know is that $P\ {}_{\geq p}\!\oplus\ Q \sqsubseteq P\ {}_{p}\!\oplus\ Q$, that 'at least p' is satisfied by 'exactly p'. The

```
if      $w = H$        $\rightarrow w{:=}\ w - 1\ \ {}_{\geq e}\!\oplus\ \ w{:=}\ w + \delta$
$\Box$  $L < w < H$    $\rightarrow w{:=}\ w + \delta\ \ {}_{\geq f}\!\oplus\ \ $fail
$\Box$      $w = L$    $\rightarrow w{:=}\ w + 1\ \ {}_{\geq e}\!\oplus\ \ w{:=}\ w + \delta$
fi;
$\{L \leq w \leq H\}$
```

Fig. 6. Regulator specification (first attempt)

specification says therefore that when the water level is dangerously high ($w = H$) the regulator will lower it (executing $w{:=}\ w - 1$, ignoring δ) with probability at least e. When the water level lies strictly between the boundaries, in the region safe from imminent catastrophe, the regulator does not need to intervene: with probability at least f it updates the water level as directed by the environment, and failure to do so is modelled by the statement **fail**, which we explain below.

To begin our analysis, we move the boundary check $\{L \leq w \leq H\}$ into the branches of the alternation. In the worst case for $w = H$ (assuming pessimistically that δ could be 1) the result is the statement

$$w{:=}\ w - 1\ {}_{\geq e}\!\oplus\ \textbf{abort}\ .$$

(The $w = L$ case is similar.)

Here we are reserving **abort** to model *catastrophic failure*: for us, that is when w escapes the boundaries. The statement **fail** on the other

[7] More precisely, we define $P\ {}_{\geq p}\!\oplus\ Q$ to be

$$P\ {}_{p}\!\oplus\ (P \sqcap Q)\ ,$$

so that an implementor must choose P with probability p; for the remaining $\overline{p}$ he may choose either P or Q.

hand is deemed a lesser failure — immediate catastrophe is not its consequence.[8] However an occurrence of **fail** would still require invocation of some back-up procedure while necessary repairs were carried out; we call that *maintenance failure*. The different levels of severity mean different reliabilities: maintenance failure would be tolerated at a much higher rate than catastrophic failure would be, and thus in general e (catastrophic reliability) would be expected to be much higher than f (maintenance reliability). In any case $\overline{e}$ should be considerably less than the failure rate of the components (see below) — that, as we will see, can be achieved by the design of the implementation.

Having distinguished between failure types, for simplicity we drop that distinction — modelling both with **abort** in our formal treatment. We do, however, preserve the brief association with the notion of 'maintenance failure' in that we allow f to be much less than e: we can tolerate maintenance failures at a higher rate. With this final simplification the regulator reduces to the program in Fig. 7.

```
$RegSpec(w,\delta)$  :=  if    $w \geq H$      $\rightarrow w := w - 1 \ {}_e\!\oplus$ abort
                         □ $L < w < H$  $\rightarrow w := w + \delta \ {}_f\!\oplus$ abort
                         □     $w \leq L$      $\rightarrow w := w + 1 \ {}_e\!\oplus$ abort
                         fi
```

We replace ${}_{\geq e}\!\oplus$ *etc.* by the simpler ${}_e\!\oplus$ without loss of generality, since if an implementation satisfies the latter it will certainly satisfy the former.

Fig. 7. Regulator specification

4.4 The Regulator Implementation

In reality a regulator would not have direct access to the water level, but rather would rely on the sensors to relay that information — it is the sensors that cause the overall failure, and our task is to minimise that risk. Thus the implementation set out in Fig. 8 does not contain w in its guards, but rather uses variables g (for 'guess', the last successful sensor reading of the water level) and i (the length of time that the sensor has continuously failed). In each time unit, the sensor fails with probability s.

[8] The **fail** occurs for example when the regulator ignores δ because an internal sensor has failed for some time — its estimate of the water level is so inaccurate that it acts as if the boundaries might be violated even when they cannot.

```
$RegImp(w,\delta)$  :=  if    $g + i \geq H$          $\rightarrow w := w - 1$
                        □  $L + i < g < H - i$  $\rightarrow w := w + \delta$
                        □     $g - i \leq L$          $\rightarrow w := w + 1$
                        fi;
                        $i := i + 1 \;\; {}_s\oplus \;\; i, g := 0, w$
```

Fig. 8. Regulator implementation

If $i = 0$ the sensor is working and the water-level reading g coincides with the actual water level w. If on the other hand $i > 0$, then g only holds the last accurate reading — for g is *not* updated along with an increase in i — so the regulator must 'guess', calculating bounds on the actual water level using the (invariant) relation

$$g - i \;\leq\; w \;\leq\; g + i\;.$$

The probabilistic assignment to i and g models a simple time-independent failure, repair rate of $1{-}s$ and s respectively. And now the extent to which the regulator can avert catastrophe by operating under the invariant above is measured by the relationship between e and s (as below).

Before we discover formally what that relationship is, arguing informally we can see that provided $g + i \geq H$ and $g - i \leq L$ do not hold simultaneously the regulator can safely push the water level away from one boundary without running the risk of crossing the other. After some time however (and in the worst case when $i > (H{-}L)/2$) that situation is no longer tenable, and at that point the regulator can only assume that the water level is both too low and too high, an impossibility that can be resolved (in the specification at least) only by abortion. Thus we would expect that e is roughly at least $1{-}s^{(H-L)/2}$, the probability that the sensor fails $(H{-}L)/2$ times in a row. As discussed above we now discover the precise quantitative relation between s, f and e by determining the conditions under which $RegSpec(w,\delta)$ is data-refined by $RegImp(w,\delta)$.

4.5 Proof of Data Refinement

As in Sec. 3, we take a supplier-oriented view of the problem — fixing the reliability $1{-}s$ of the sensor in the implementation, we wish to find the greatest values for e and f which will allow a data refinement. Recall that overall reliability of the regulator depends on the reliability of the sensors, and establishing a data refinement is contingent on that dependency.

We prove the refinement in two stages: starting from the specification in Fig. 7 we first add the variable i, to reach the intermediate program set out in Fig. 9; only then do we introduce the variable g. We sketch the details of the formal proofs here, as well as appealing freely to standard data refinement results — for example that the relation of data refinement is transitive, and indeed that ordinary program refinement is simply a special case [2].

```
if   $w \geq H$       $\rightarrow \{i < (H-L)/2\};\ w{:=}\ w-1$
□ $L < w < H$ $\rightarrow \{i = 0\};\ w{:=}\ w+\delta$
□   $w \leq L$       $\rightarrow \{i < (H-L)/2\};\ w{:=}\ w+1$
fi;
$i{:=}\ i+1\ {}_s\!\oplus\ i{:=}\ 0$
```

Fig. 9. Intermediate refinement step

Lemma 4.1. *RegSpec*(w,δ) is data-refined by the datatype set out in Fig. 9 provided that f is no greater than $1-s$ and that e is no greater than the solution of (6), repeated here:

$$x^N \quad = \quad \overline{c}(x^{N-1} + cx^{N-2} + ... + c^{N-2}x + c^{N-1}) \ .$$

Proof: The lemma follows by noting first that each guarded statement, together with the assignment to i that follows, is essentially an instance of the faulty skipper in Fig. 1. To prove a data refinement, we use a representation function

$$rep \quad := \quad (\|n : 0 \leq n \leq (H-L)/2 : i{:=}\ n \ @ \ p_n);\ i{:\leq}\ i \qquad (7)$$

where the p_n are defined by (5) of Sec. 3. The required data refinement can now be proved via (7): *rep* distributes through the guards (they do not involve i and *rep* does not mention w); moreover despite the p_n being solutions to the faulty N-**skip**per when $N = (H-L)/2$ (thus applicable to the boundaries) we note that the nondeterministic decrease of i in (7) gives us the refinements (2) for the intermediate $L < w < H$ case (corresponding to a faulty 2-**skip**per) as well. The details are routine and are omitted. □

Lemma 4.2. The datatype in Fig. 9 is data refined by the *RegImp*(w,δ) of Fig. 8.

Proof: We add the variable g to the intermediate datatype in Fig. 9 using the standard technique of coupling invariants [2]. We observe first that the real water level w is always maintained between the estimated boundaries $g - i$ and $g + i$, hence we define the coupling invariant

$$I \quad := \quad g - i \leq w \leq g + i \ .$$

Using this, we can (routinely) replace the guards and augment the assignment; again we omit the details.

□

With Lem. 4.1 and Lem. 4.2 we can finally conclude that using a sensor of reliability $(1-s)$ guarantees overall system reliability of at least (and in fact strictly greater than) $1-s^{(H-L)/2}$ — this of course is reliability with respect to ultimate catastrophe.

5 Conclusion

In this paper we have made two contributions: we illustrated how data refinements within a probabilistic domain may be proved using the technique of simulations; and secondly we argued that establishing probabilistic data refinements provides a quantitative link between system and component level reliabilities.

It is well known that representation programs (our *rep*) are not necessarily unique, and one might conjecture that a non-probabilistic representation might be found that would satisfy the refinements in (2) and thus avoid the calculations presented here. That the probabilistic nature of our example needs the extended techniques can be seen by considering the 2-**skip**per — for we may assume a finite state space and thus only finitely many possibilities (9 in all) for a non-probabilistic *rep*, none of which satisfy the refinements in (2).

More generally we can explain the choice of *rep* by considering how accurately an observer can guess the value of the local variable at 'run-time' of a program using the 2-**skip**per. We assume that he knows the implementation's code, but not the *actual* run-time value of i (since he cannot observe i directly). After several calls to *opc* at best he can only infer a probability distribution for i; from this he can calculate the chance that the 2-**skip**per will abort next time *opc* is called — it is the probability that the actual value of i is 2. Hence *rep* encodes the observer's most accurate knowledge of i's value — if *rep* was non-probabilistic that would correspond either to perfect knowledge (for a deterministic *rep*) or to complete lack of knowledge (for a nondeterministic *rep*); in neither case is this appropriate for the 2-**skip**per as the observer's knowledge lies somewhere in between and can only be described by a probability distribution. More details concerning the general treatment of probabilistic assignments to local variables can be found elsewhere [7].

We hope to extend the use of the probabilistic model presented here to the application of quantified analyses of safety critical systems in general. For example our modelling all types of system failures as **abort** is certainly unacceptable for a quantified assessment of risk for which we would need to continue to distinguish between the possible identified hazards (for risk measures the cost modified by likelihood of all the hazards [14]). Building a sufficiently simple model where we can make such distinctions is a topic for future research.

Acknowledgements

This paper reports work carried out at Oxford within a project supported by the *EPSRC*, directed jointly by Jeff Sanders and Carroll Morgan; and we are grateful for Sanders' contribution to this topic.

References

1. J.-R. Abrial. *The B Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.
2. C.C.Morgan. Auxiliary variables in data refinement. *Information Processing Letters*, 29(6):293–296, December 1988. Reprinted in [12].
3. E.W. Dijkstra. *A Discipline of Programming.* Prentice Hall International, Englewood Cliffs, N.J., 1976.
4. P. H. B. Gardiner and C. C. Morgan. Data refinement of predicate transformers. *Theoretical Computer Science*, 87:143–162, 1991. Reprinted in [12].
5. Paul Gardiner and Carroll Morgan. A single complete rule for data refinement. *Formal Aspects of Computing*, 5(4):367–382, 1993. Reprinted in [12].
6. G. Grimmett and D. Welsh. *Probability: an Introduction.* Oxford Science Publications, 1986.
7. Probabilistic Systems Group. A quantified measure of security 1: a relational model. Available via *http* [13].
8. Jifeng He, K.Seidel, and A. K. McIver. Probabilistic models for the guarded command language. *Science of Computer Programming*, 28(2,3):171–192, January 1997.
9. C.A.R Hoare, Jifeng He, and J.W. Sanders. Prespecification in data refinement. *Information Processing Letters*, 25(2), May 1987.
10. C.B. Jones. *Systematic Software Development using VDM.* 2ed, Prentice-Hall, 1990.

11. C. C. Morgan, A. K. McIver, and K. Seidel. Probabilistic predicate transformers. *ACM Transactions on Programming Languages and Systems*, 18(3):325–353, May 1996.
12. C. C. Morgan and T. N. Vickers, editors. *On the Refinement Calculus.* FACIT Series in Computer Science. Springer-Verlag, Berlin, 1994.
13. PSG. Probabilistic Systems Group: Collected reports. *http://www.comlab.ox.ac.uk/oucl/groups/probs/bibliography.html.*
14. N. Storey. *Safety-critical computer systems.* Addison-Wesley, 1996.

Appendix: Some algebraic laws of probabilistic programs

1. $(P \ {}_{q}\!\oplus\ Q); R \quad = \quad P; R \ {}_{p}\!\oplus\ Q; R$
2. $R; (P \ {}_{p}\!\oplus\ Q) \quad \sqsubseteq \quad R; P \ {}_{p}\!\oplus\ R; Q$

and if R is a deterministic and standard program,

3. $R; (P \ {}_{p}\!\oplus\ Q) \quad = \quad R; P \ {}_{p}\!\oplus\ R; Q$.