JFP **30**, e12, 41 pages, 2020. © NASA 2020. Published by Cambridge University Press. This is a work of the US Government and is not subject to copyright protection within the United States. Published by Cambridge University Press.
doi:10.1017/S0956796820000118

Fault-tolerant functional reactive programming (extended version)

IVAN PEREZ
National Institute of Aerospace, Hampton, VA, 23666, USA
(*e-mail:* ivan.perez@nianet.org)

ALWYN E. GOODLOE
NASA Langley Research Center, Hampton, VA, 23681, USA
(*e-mail:* a.goodloe@nasa.gov)

Abstract

Highly critical application domains, like medicine and aerospace, require the use of strict design, implementation, and validation techniques. Functional languages have been used in these domains to develop synchronous dataflow programming languages for reactive systems. Causal stream functions and functional reactive programming (FRP) capture the essence of those languages in a way that is both elegant and robust. To guarantee that critical systems can operate under high stress over long periods of time, these applications require clear specifications of possible faults and hazards, and how they are being handled. Modeling failure is straightforward in functional languages, and many functional reactive abstractions incorporate support for failure or termination. However, handling *unknown types of faults*, and incorporating *fault tolerance* into FRP, requires a different construction and remains an open problem. This work demonstrates how to extend an existing functional reactive framework with fault tolerance features. At value level, we tag faulty signals with reliability and probability information and use random testing to inject faults and validate system properties encoded in temporal logic. At type level, we tag components with the kinds of faults they may exhibit and use type-level programming to obtain compile-time guarantees of key aspects of fault tolerance. Our approach is powerful enough to be used in systems with realistic complexity, and flexible enough to be used to guide system analysis and design, validate system properties in the presence of faults, perform runtime monitoring, and study the effects of different fault tolerance mechanisms.

1 Introduction

Mission critical systems – those in which a malfunction may result in loss of life or great economic impact – require the use of careful design, implementation, testing, and deployment techniques. In domains like aviation and transportation, the development of both hardware and software is heavily regulated and strict guidelines must be followed (RTCA, 2011).

The use of synchronous programming languages with clear semantics is frequent in these domains (Halbwachs *et al.*, 1991; Berry *et al.*, 2000; Dormoy, 2008). These languages are normally based on a notion of streams and stream functions, and they must guarantee, at compile time, that all streams are causal and well formed.

The essence of these languages can be captured via causal stream functions and some forms of functional reactive programming (FRP) (Elliott & Hudak, 1997; Nilsson *et al.*, 2002; Courtney *et al.*, 2003). While FRP has traditionally been applied to domains like interactive programming and games, abstractions like monadic stream functions (MSFs) (Perez *et al.*, 2016; Perez, 2017; Bärenz & Perez, 2018) help bridge the gap between discrete-time causal stream programming and continuous-time FRP.

However, even when functional reactive and synchronous dataflow systems are implemented according to specifications, they may fail in production due to undetected software bugs in other components, hardware failures, or environmental hazards. It is extremely important to determine the most likely hardware and software faults and environmental damage and to minimize their impact by introducing mechanisms for *fault tolerance* (Avižienis, 1967, 1976; Butler, 2008).

A general way of capturing and addressing faults in typed functional languages is by means of optional values or values that encode errors, and possibly the reasons behind those errors. FRP frameworks like MSFs and Yampa introduce notions of failure or termination via the use of *Maybe* and *Either*.

While these types capture notions of *detectable failures*, they are not suitable to represent *failures that are not detectable* or *correctable* at a given stage. For example, in space applications, we normally see bit flips due to radiation, which alter values in memory (Bedingfield *et al.*, 1996). Without further adjustments to our program, we may not be able to tell that an error occurred, let alone correct it.

This paper presents mechanisms to encode the possibility of undetectable failures in a functional specification of a reactive system. We do so in three ways: by tagging values with different degrees of confidence, by means of distributions denoting possible values and their probabilities, and by tagging values with value-level and type-level fault sets that denote the kinds of failures that may have affected them. We combine this approach with random testing encoding desired system properties using temporal logic, for a more systematic and comprehensive verification and validation methodology.

This extra information serves two purposes: during system design and implementation, it helps understand the possible failures, how they may affect system requirements, and what fault tolerance mechanisms need to be introduced to address those faults. During execution, it helps understand the confidence we can place on a specific result and make dynamic adjustments and decisions about a mission based on the margins of error we can tolerate. We discuss prototypes in Haskell and in Idris, based on an extension of MSFs to work with parameterized monads, and explain which kinds of analysis are possible in each language.

Our work makes the following contributions:

- We present a polymorphic type constructor to encode *potentially unreliable data*, show how it can be used to compute the expected reliability or availability of reactive constructs (Section 3), and show how to introduce fault tolerance mechanisms to increase the overall reliability (Section 4).
- We provide a representation of certainty based on *probability* distributions and show how to calculate the total confidence of a reactive network (Section 5).

- We demonstrate how to tag components with the faults that can affect them, to synthesize an enumeration of all potential faults in a reactive system (Section 6).
- We demonstrate how to use QuickCheck to validate temporal logic properties of a system, *injecting faults* with different probabilities (Section 7).
- We extend MSFs, an existing formalism for FRP that combines Arrows with Monads, with support for *parameterized* or *indexed monads*, which facilitates chaining constructions working on different monads (Section 8).
- We show how to use *type-level programming* to capture *potential faults*, leveraging fault analysis on the type system to obtain a proof of the faults that may affect the behavior of a reactive network (Section 9).

This paper is an extended version of Perez (2018). Section 7 is an original contribution in this extended version, with examples from Perez *et al.* (2019).

In the following section, we introduce basic concepts in FRP and fault tolerance and illustrate the problem we seek to address.

2 Background

In the interest of making this paper sufficiently self-contained, we summarize the basics of FRP and MSFs in the following. We later describe basic ideas from the field of fault tolerance, together with an example that illustrates the problems we seek to address. For further details, see earlier papers on FRP (Elliott & Hudak, 1997; Nilsson *et al.*, 2002; Courtney *et al.*, 2003) and MSFs (Perez *et al.*, 2016; Perez, 2017; Bärenz & Perez, 2018). This presentation draws heavily from the summaries in Perez & Nilsson, (2017) and Courtney *et al.* (2003). For details on fault tolerance, see Avižienis (1967, 1976) and, for an in-depth introduction, see Butler (2008).

2.1 Functional reactive programming

FRP is a programming paradigm to describe hybrid systems that operate on time-varying data. FRP is structured around the concept of *signal*, which conceptually can be seen as a function from time to values of some type:

$$\mathit{Signal}\ \alpha \approx \mathit{Time} \rightarrow \alpha$$

Time is (notionally) continuous and represented as a nonnegative real number. The type parameter α specifies the type of values carried by the signal. For example, the type of an animation would be *Signal Picture* for some type *Picture* representing static pictures. Signals can also represent input data, like values read from a sensor.

Additional constraints are required to make this abstraction executable. First, it is necessary to limit how much of the history of a signal can be examined, to avoid memory leaks. Second, if we are interested in running signals in real time, we require them to be *causal*: they cannot depend on other signals at future times. FRP implementations address these concerns by limiting the ability to sample signals at arbitrary points in time. Also, although FRP is conceptually continuous, implementations still execute by sampling inputs at discrete points in time and some even hide time completely.

The space of FRP frameworks can be subdivided into two main branches, namely Classic FRP (Elliott & Hudak, 1997) and Arrowized FRP (Nilsson *et al.*, 2002). Classic FRP programs are structured around signals or a similar notion representing internal and external time-varying data. In contrast, Arrowized FRP programs are defined using causal functions between signals, or *signal functions*, connected to the outside world only at the top level.

To address some of the limitations of different forms of FRP and bridge the gap between Classic and Arrowized FRP, and between continuous-time and discrete-time variants, MSFs separate the notion of time from the notion of causal transformation over a varying sampled input. MSFs can be empowered with additional features by means of different monads, which can serve both to make them more expressive, to synthesize information, or with different control mechanisms. In the following, we turn to MSFs and later explain current limitations that this paper addresses.

2.2 Monadic stream functions

MSFs are an abstraction that can be used to implement FRP and supports discrete and continuous time, and both Classic and Arrowized FRP[1]. MSFs are defined by a polymorphic type *MSF* and a function *step* that applies an *MSF* to an input and returns, in a monadic context, an output and a continuation:

$$\textbf{newtype}\ \mathit{MSF}\ m\ a\ b$$

$$\mathit{step} :: \mathit{Monad}\ m \Rightarrow \mathit{MSF}\ m\ a\ b \to a \to m\ (b, \mathit{MSF}\ m\ a\ b)$$

We purposefully hide the details of the definition of *MSF*. Functions to define and combine MSFs, while preserving causality and avoiding leaks, will be provided in the rest of this section.

The type *MSF* and the *step* function alone do not represent causal functions on streams. It is only when we successively apply the function to a stream of inputs and consume the side effects that we get the unrolled, streamed version of the function. Causality, or the requirement that the n-th element of the output stream only depend on the first n elements of the input stream, is obtained as a consequence of applying the MSF continuations *step by step*, or sample by sample. For the purposes of exposition, we use the following function to apply an MSF to a *finite list* of inputs, with effects and continuations chained sequentially. This is merely a debugging aid, not how MSFs are actually executed:

$$\mathit{embed} :: \mathit{Monad}\ m \Rightarrow \mathit{MSF}\ m\ a\ b \to [a] \to m\ [b]$$

MSFs are *Arrows* (Hughes, 2000), and so *Arrow* combinators can be used to define MSFs compositionally (Bärenz *et al.*, 2016). Some central combinators are *arr* that lifts an ordinary function to a stateless signal function, composition $\ggg$, parallel composition &&&. Through the use of these and related combinators, arbitrary MSF networks can be expressed. Specialized for MSFs, the basic *Arrow* operations have the following types:

[1] The Haskell packages `dunai` and `bearriver` implement, respectively, MSFs and Arrowized FRP on top of MSFs.

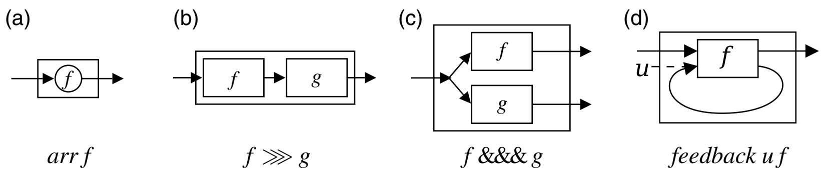

Fig. 1. Basic MSF combinators.

$$\begin{aligned}
&\mathit{arr} &&:: (a \to b) &&\to \mathit{MSF}\ m\ a\ b\\
&(\ggg) &&:: \mathit{MSF}\ m\ a\ b \to \mathit{MSF}\ m\ b\ c &&\to \mathit{MSF}\ m\ a\ c\\
&(\&\&\&) &&:: \mathit{MSF}\ m\ a\ b \to \mathit{MSF}\ m\ a\ c &&\to \mathit{MSF}\ m\ a\ (b, c)
\end{aligned}$$

MSFs must remain leak-free, so we introduce limited ways of depending on past values. To keep state by producing an extra *accumulator* accessible in future iterations we use

$$\mathit{feedback} :: c \to \mathit{MSF}\ m\ (a, c)\ (b, c) \to \mathit{MSF}\ m\ a\ b$$

This combinator takes an initial value for the accumulator, runs the MSF, and feeds the new accumulator back for future iterations.

We can think of streams and MSFs using a simple flow chart analogy (Figure 1). Lines or “wires” represent streams, with arrowheads indicating the direction of flow. Boxes represent MSFs, with one input flowing into the box’s input port and an output stream flowing out of the box’s output port.

Example. A counter, for example, can be defined as follows. The second MSF in this expression calculates the cumulative sum of its inputs, initializing an accumulator to 0 and using a feedback loop:

```
count :: (Num n, Monad m) ⇒ MSF m () n
count = arr (const 1) ⋙ feedback 0 (arr add2)
  where
    add2 (n, acc) = let n′ = n + acc in (n′, n′)
```

2.2.1 Monads

MSFs can be combined with different monads for different effects. We provide a general function *arrM* to lift a Kleisli arrow, so that it is applied pointwise to every sample:

$$\mathit{arrM} :: \mathit{Monad}\ m \Rightarrow (a \to m\ b) \to \mathit{MSF}\ m\ a\ b$$

The use of monads with MSFs provides great versatility. For example, we can make certain values available in an environment in a *Reader* monad, without having to route them down manually as inputs to other MSFs:

```
data Env = Env {windowWidth  :: Int
               , windowHeight :: Int
               }
rotateTapPos180 :: MSF (Reader Env) (Int, Int) (Int, Int)
rotateTapPos180 = proc (x, y) → do
```

$$\begin{aligned}
&\mathit{winW} \leftarrow \mathit{arrM}\ (\backslash\_ \rightarrow \mathit{asks\ windowWidth})\ \prec ()\\
&\mathit{winH}\ \leftarrow \mathit{arrM}\ (\backslash\_ \rightarrow \mathit{asks\ windowHeight}) \prec ()\\
&\mathit{returnA} \prec (\mathit{winW} - x, \mathit{winH} - y)
\end{aligned}$$

It is possible to “flatten” an MSF by removing the monadic effect, by means of what are called *MSF running functions*. This normally requires extra inputs, extra outputs, or extra continuations. For example, the running function for the Reader monad has signature:

$$\mathit{runReaderS\_} :: \mathit{MSF}\ (\mathit{ReaderT}\ r\ m)\ a\ b \rightarrow r \rightarrow \mathit{MSF}\ m\ a\ b$$

We can test *rotateTapPos180* in a session, by providing an additional environment, plus the input samples denoting examples of positions where the user could have tapped on the screen. Because *Reader* is defined as a *ReaderT* on the *Identity* monad, we use *runIdentity* to extract the value from the bottom monad in the stack:

$$\begin{aligned}
&\mathit{ghci} > \mathit{runIdentity}\ \$\\
&\quad \mathit{embed}\ (\mathit{runReaderS\_}\ (\mathit{rotateMousePos180})\ (\mathit{Env}\ 1024\ 768))\ [(10, 10), (100, 100)]\\
&[(1014, 758), (924, 668)]
\end{aligned}$$

Analogously, the associated execution function for terminating *MSF*s would have type:

$$\mathit{runMaybeS} :: \mathit{Monad}\ m \Rightarrow \mathit{MSF}\ (\mathit{MaybeT}\ m)\ a\ b \rightarrow \mathit{MSF}\ m\ a\ (\mathit{Maybe}\ b)$$

For this monad, *step* has type $\mathit{MSF\ Maybe}\ a\ b \rightarrow a \rightarrow \mathit{Maybe}\ (b, \mathit{MSF\ Maybe}\ a\ b)$ and may produce *no continuation* (*runMaybeS* applied to such an *MSF* would output *Nothing* from that point on). “Recovering” from failure requires an additional continuation:

$$\mathit{catchM} :: \mathit{Monad}\ m \Rightarrow \mathit{MSF}\ (\mathit{MaybeT}\ m)\ a\ b \rightarrow \mathit{MSF}\ m\ a\ b \rightarrow \mathit{MSF}\ m\ a\ b$$

With a monad *Either c* for some type *c*, the recovering function resembles Yampa’s *switch* combinator, showing that switching emerges for free by combining MSFs with the *Either* monad. When combined with the list monad, MSFs give rise to *dynamic collections* of MSFs in *parallel*.

The extensibility provided by different monads, transformers and their running functions, makes MSFs unusually flexible for describing dynamic reactive systems. Different choices of inputs, outputs, and monads lets us realize streams, monadic streams, Classic FRP signals and Arrowized FRP signal functions, all with the same abstraction. MSFs that do not depend on their input produce a Monadic Stream (a stream of outputs in a monadic context), or an ordinary stream if the monad is identity. Monadic Streams with a Reader monad that includes external inputs and/or time can implement Classic FRP. Introducing time via a Reader monad in an MSF can be used to implement Arrowized FRP, obtaining a step function that is isomorphic to Yampa’s internal representation of initialized signal functions. This makes it possible to run Yampa simulations on top of an intermediate layer that implements an API-compatible version of Yampa on top of MSFs.

2.3 Faults in reactive systems

Let us illustrate the issues we seek to address with an example. Imagine that we are building a system to control a spacecraft. To move and orient the spacecraft, we carry out a phase of *attitude determination* (determining the spacecraft's position and direction) prior to *attitude control* (correcting position and direction via actuators). Errors in these calculations can lead to the spacecraft deviating from its path, being lost, or colliding.

To determine the attitude, we obtain data from a *star tracker*, which estimates the position and orientation relative to stars whose locations in space we know, and from an *inertial measurement unit* (IMU), which combines gyroscopes and accelerometers to estimate the spacecraft's acceleration, linear velocity, orientation, and surrounding magnetic fields.

A schematic definition of our control system follows. Our reactive control *MSF* takes a desired attitude as input and produces a set of actions as output. Internally, it gathers data from a star tracker and the IMU. The use of feedback allows us to calculate the new attitude based on the last known attitude and the information gathered from the sensors:

```
controlSystem :: Attitude → MSF m Attitude [Action]
controlSystem initialAttitude = proc (desiredAttitude) → do
  stars       ← starTrackerSense ⤙ ()
  inertialInfo ← imuSense        ⤙ ()
  attitude    ← feedback initialAttitude (arr (dup ∘ calculateAttitude))
                   ⤙ (stars, inertialInfo)
  let actions = calculateActions desiredAttitude attitude

  returnA ⤙ actions

  -- Predefined elsewhere
type Action      = ...
type Attitude    = ...
type StarInfo    = ...
type InertialInfo = ...

starTrackerSense :: MSF m () [StarInfo]
imuSense         :: MSF m () InertialInfo
calculateAttitude :: (([StarInfo], InertialInfo), Attitude) → Attitude
calculateActions :: Attitude → Attitude → [Action]
dup              :: a → (a, a)
```

Ideally, we would assume that the hardware works perfectly that values from sensors are accurate and that the memory is never corrupted. However, this is not true in practice, and critical systems that carry out important operations over long periods of time in hostile environments without maintenance, like a satellite, warrant extra levels of reliability.

In particular, in our example, nothing tells us *whether data is inaccurate*, *how inaccurate it is*, and what *kinds of faults* may have lead to incorrect results. In cases like these, using a *Maybe* type to encode the possibility of failure in the output type of a sensor is not enough.

The use of techniques to minimize the impact of failures is grouped under the umbrella term of *fault tolerance* (Avižienis, 1967, 1976). In fault tolerance, redundancy can be used to compare results from multiple units and disable those that fail or to average them and lower the impact of minor deviations (*voting*). For instance, in our example, three star trackers could be used to calculate a midpoint, as well as to detect if either star tracker is providing values that are too far from the average. Additionally, and to limit the possibility of redundant units failing for the same reason, subsystems can be physically and electrically isolated, placing them in separate *fault containment regions (FCRs)*.

The combination of these fault tolerance mechanisms helps determine which of possible faults may still affect system operations and defines the *fault model*. Total reliability is never possible, so our fault model is likely to determine both the kinds of faults that are handled and also how many simultaneous faults may be tolerated. For instance, a system may be able to deal with a value not being available, but not be with a value being incorrect. Others are capable of handling Byzantine errors (those in which different subsystems have different incorrect values for the same conceptual element), whereas others may only be able to do so if the error is corrected before another simultaneous error occurs.

2.4 Limitations

FRP and MSF definitions normally assume that systems do not fail unexpectedly and all errors can be detected, which is not always true. The limitations we seek to address are[2]:

Availability. System descriptions using MSFs do not help us understand how reliable systems can be and how much time we can expect them to be un/available. While we may have obtained information about the expected availability or reliability of subsystems or components from the manufacturer or from prior experience, these specifications hide data dependencies and do not help synthesize a global reliability factor based on those of expected components. This also makes it hard to determine if fault tolerance is needed and whether it would guarantee system requirements in the presence of faults.

Ranges and probabilities. These system descriptions operate with exact values, and, unless randomized testing is used, simulations would normally work with ideal inputs from the sensors. In practice, most components and subsystems have margins of error, and a range of values may be expected. The use of probability distributions constitutes a better mechanism to represent possible input and, potentially, analyze the possibility of values being within the desired margins of tolerance.

Fault model. These frameworks let us specify data dependencies from multiple subsystems and therefore hide details in the implementation. In particular, nothing in our example lets us know that attitude calculation may be affected by, for example, condensation in the star tracker's battery unit. The use of monads, available in MSFs, opens an opportunity to specify these failures at the type level and let them propagate across a network, tagging all reactive constructs that may have been affected.

In the following, we present an approach to model, validate, and verify reactive systems when results are unreliable and different subsystems can fail. We do so in two ways. First, we alter values to attach reliability information or to produce multiple values with different probabilities. This information is dynamic and available at runtime, and so validation

[2] In the following, when we refer to MSFs, the limitations apply generally also to other forms of FRP and stream programming.

Table 1. Summary of fault information introduced in reactive systems, level at which it is introduced, the type chosen to represent it, and the sections where it is discussed

Information	Level	Representation	Sections
Reliability	Value	Fault tolerance/Writer monad, with numeric values	Sections 3, 4
Probabilities	Value	Dist (probability) monad	Section 5
Fault sets	Value	Fault tolerance/Writer monad, with fault sets	Sections 6, 7
Fault sets	Type	Tagging monad, with type-level fault sets	Sections 8, 9

of these systems is performed using temporal logic and QuickCheck. Second, we elevate value-level information to type level, to leverage the analysis of possible faults on the type system. To do so, we extend MSFs to operate on parameterized or indexed monads. Because fault information is expressed at the type level and performed by the compiler following typing rules, this constitutes a form of verification that provided that the characterization of faults for different subcomponents is performed correctly. Our presentation is based on MSFs, as described in Section 2, and relies on the choice of the right monad to introduce fault tolerance information (Table 1). Implementations in Haskell and in Idris are discussed.

3 Reliability factors

Ideally, we would like to be able to estimate the reliability of a complex system based on the reliability of the subsystems that form it, which may itself be obtained from the manufacturer or from previous experiments.

Let us present an example to understand how to introduce reliability or confidence factors in reactive MSFs. Imagine that we gather data from the two sensors used in the example in Section 2 and want to tag them with a reliability factor. For the sake of simplicity, we use a numerical value between 0 and 1, with 0 expressing *no confidence or reliability* on the accuracy of the value, and 1 expressing *absolute certainty*. In Section 5, we show how to use more structured representations of reliability.

We can represent this information in MSFs by just changing the inputs and the outputs. First, let us introduce a type for values with a degree of uncertainty:

$$\begin{aligned}&\textbf{type}\ \mathit{Uncertain}\ a = (\mathit{Certainty}, a)\\ &\textbf{type}\ \mathit{Certainty} \quad = \mathit{Double} \quad \text{-- zero to one}\end{aligned}$$

For simplicity, we assume that reliability is constant and irrespective of the conditions of operation (we later present how reliability could be dynamically changed). We rely on *MSF*s defined earlier and state that both sensors have a 90% accuracy:

$$\begin{aligned}&\mathit{starTrackerSense}' :: \mathit{MSF}\ m\ ()\ (\mathit{Uncertain}\ [\mathit{StarInfo}])\\ &\mathit{starTrackerSense}' = \mathit{starTrackerSense} \ggg \mathit{arr}\ (\lambda \mathit{starData} \rightarrow (0.9, \mathit{starData}))\\ &\mathit{imuSense}' :: \mathit{MSF}\ m\ ()\ (\mathit{Uncertain}\ \mathit{InertialInfo})\\ &\mathit{imuSense}' = \mathit{imuSense} \ggg \mathit{arr}\ (\lambda \mathit{imuData} \rightarrow (0.9, \mathit{imuData}))\end{aligned}$$

These new *MSF*s add some extra information to the output. We now adapt both the step function and the attitude and action calculation functions to handle uncertainty. Functions like *controlSystemStep′* just route the uncertainty together with the accompanying values:

```
controlSystemStep′ :: Uncertain Attitude → MSF m Attitude (Uncertain [Action])
controlSystemStep′ initialAttitude = proc (desiredAttitude) → do
  stars        ← starTrackerSense′ ⤙ ()
  inertialInfo ← imuSense′         ⤙ ()
  attitude     ← feedback initialAttitude (arr (dup ∘ calculateAttitude′))
                   ⤙ (stars, inertialInfo)
  let actions  = calculateActions′ desiredAttitude attitude
  returnA ⤙ actions
```

Functions like *calculateAttitude′*, however, need access to this factor to calculate the combined uncertainty of the result. If the sensors are independent (e.g., one does not become less accurate when the other fails to report accurate data), then we could say that the result is accurate if both sensors are accurate. Therefore, the combined confidence of a result based on data from two sensors is the multiplication of the reliabilities of both. This operates similar to probabilities of independent events.

```
calculateAttitude′ ::  ((Uncertain [StarInfo], Uncertain InertialInfo), Uncertain Attitude)
                     → Uncertain Attitude
calculateAttitude′ ((starUnc, starData), (inerUnc, inerData)), (attUnc, attData)) =
  let attitude = calculateAttitude ((starData, inerData), attData)
  in  (attUnc ∗ starUnc ∗ inerUnc, attitude)
calculateActions′  :: Attitude → Uncertain Attitude → Uncertain [Action]
calculateActions′ desiredAttitude (attUnc, attData) =
  (attUnc, (calculateActions desiredAttitude attData))
```

Our main function *controlSystemStep′* now also estimates the confidence on the action. In a hypothetical system, if we execute this action and obtain a value lower than a given threshold, we might want to use an alternative calculation, turn off some devices, or address the loss of accuracy in some other way.

Manipulating factors by hand is error-prone. Without a facility to guarantee that calculations are correct, this method does not increase the knowledge we have about certainty in our system, and a false sense of security is placed on potentially incorrect results.

We can abstract the underlying details into a monad that performs the calculation steps from the reliability factors. This monad is exactly like a *Writer* monad, in which the monoid are real numbers between 0 and 1 in the log, using multiplication as the monoidal operation and 1 as identity. However, monoidal operations would only maintain or decrease certainty (i.e., multiplication of a positive value by a value between 0 and 1 can render it smaller, but not larger). To provide fault tolerance, we also need a way to *increase* confidence on our system, which requires that we inspect the contents of the *Writer*.

The following section captures this pattern of operation with a custom monad, which we later enrich with further structure to introduce fault tolerance mechanisms.

3.1 Fault tolerance monad

Let us abstract the details of the previous example into a type for “uncertain” or “imprecise” values. As we saw before, this is just a polymorphic product type *FaultTolerance e a*, in which we tag values of type *a* with extra reliability information of type *e*. In order to operate with these values, we first need to be able to combine fault information, so they must have a binary operation with identity, making them a *monoid*. This structure is equivalent to a *Writer* monad, making it a *Functor*, an *Applicative*, and a *Monad*. The use of the functor operation *fmap* leaves the reliability unchanged, while the applicative and monadic operators combine the information available using the monoidal operation *mappend* (in our example, multiplication).

To hide details from the user, we add an operation to put a value with a specific tag:

$$
\begin{aligned}
&\mathit{liftUnreliable} :: e \to a \to \mathit{FaultTolerance}\ e\ a\\
&\mathit{liftUnreliable}\ e\ x = \mathit{FaultTolerance}\ e\ x
\end{aligned}
$$

3.2 Example

Adapting the previous example to this new interface should be straightforward, and it only makes the specification simpler (we overload names where appropriate):

$$
\begin{aligned}
&\mathit{starTrackerSense'} :: \mathit{MSF}\ (\mathit{FaultTolerance}\ \mathit{Certainty})\ ()\ [\mathit{StarInfo}]\\
&\mathit{starTrackerSense'} = \mathit{starTrackerSense} \ggg \mathit{arrM}\ (\mathit{liftUnreliable}\ 0.9)\\
&\mathit{imuSense'} :: \mathit{MSF}\ (\mathit{FaultTolerance}\ \mathit{Certainty})\ ()\ \mathit{InertialInfo}\\
&\mathit{imuSense'} = \mathit{imuSense} \ggg \mathit{arrM}\ (\mathit{liftUnreliable}\ 0.9)
\end{aligned}
$$

We adapt the main control functions as follows. For the sake of simplicity, we temporarily eliminate the monad *m* from the stack. We remedy this situation later with the introduction of a monad transformer for fault tolerance.

```
controlSystemStep' :: Attitude
                        → MSF (FaultTolerance Certainty) Attitude [Action]
controlSystemStep' initialAttitude = proc (desiredAttitude) → do
  stars       ← starTrackerSense' ≺ ()
  inertialInfo ← imuSense'        ≺ ()
  attitude    ← feedback initialAttitude (arr (dup ∘ calculateAttitude))
                   ≺ (stars, inertialInfo)
  actions     ← arr (uncurry calculateActions) ≺ (desiredAttitude, attitude)
  returnA ≺ actions
```

Note that this only changes the signature of the function, but not the definition. This is a key strength of this approach, and we explore this benefit further in future sections. Calculating the attitude and the actions now becomes trivially simple and we can use the original functions, as it is leveraged onto the *Monad* instance of *FaultTolerance*.

While the example above introduces a fixed reliability, the reliability of a component might depend on different factors, including, for example, the mission runtime. For example, if we assumed a discrete notion of time, we can model the running time using a counter, which is trivially implementable using the *feedback* primitive in our language, as $counter = feedback\ 1.0\ (arr\ (dup \circ (+1.0) \circ snd))$. We could change the expression that introduced the reliability to dynamically decrease over time, for example, starting at 0.9 and decreasing to 0.5, with the calculation:

```
starTrackerSense'' :: MSF (FaultTolerance Certainty) () [StarInfo]
starTrackerSense'' = starTrackerSense ⋙ reliabilityST''

reliabilityST'' :: MSF (FaultTolerance Certainty) a a
reliabilityST'' = proc (a) → do
  time ← counter ⤙ ()
  let reliability = 0.5 + 0.4 / time
  arrM (uncurry liftUnreliable) ⤙ (reliability, a)
```

The introduction of these mechanisms makes networks produce a dynamic degree of reliability. Present reliability is no longer indicative of future reliability. This both makes the mechanism provided more powerful and versatile and also requires more thorough evaluation during validation and verification.

3.3 *Fault tolerance monad transformer*

We can generalize the previous type into a monad transformer (Liang *et al.*, 1995), to combine it with other monads. The new polymorphic type definition simply encapsulates the fault-tolerant value in a monad, which we can define like a *WriterT* transformer:

```
data FaultToleranceT e m a = FaultToleranceT
  {runFaultToleranceT :: m (e, a)}
```

As is usual with transformers, **type** *FaultTolerance* = *FaultToleranceT Identity*. Due to the similarity with the previous *FaultTolerance*, as well as the *Writer* monad and the *WriterT* monad transformer, we obviate the details of the implementations of the *Functor*, *Applicative*, and *Monad* instances. We provide two convenience functions to create values with limited and with absolute confidence:

```
liftUnreliableT :: Monoid e → m a → FaultToleranceT e m a
liftUnreliableT f m = FaultToleranceT <$> (λx → (f, x)) <$> m

liftReliableT :: Monoid e ⇒ m a → FaultToleranceT e m a
liftReliableT m = FaultToleranceT <$> (λx → (mempty, x)) <$> m
```

3.4 *Example*

Our main MSF remains largely the same, except that we must use the new operations in our calculations:

```
starTrackerSense' :: MSF (FaultToleranceT Certainty m) () [StarInfo]
starTrackerSense' = starTrackerSense ⋙ arrM (liftUnreliableT 0.9)

imuSense' :: MSF (FaultToleranceT Certainty m) InertialInfo
imuSense' = imuSense ⋙ arrM (liftUnreliableT 0.9)
```

We now simplify the main control function:

```
controlSystemStep' :: Attitude
                   → MSF (FaultToleranceT Certainty m) Attitude [Action]
controlSystemStep' initialAttitude = proc (desiredAttitude) → do
  stars       ← starTrackerSense' ─≺ ()
  inertialInfo ← imuSense'        ─≺ ()
  attitude    ← feedback initialAttitude (arr (dup ∘ calculateAttitude))
                  ─≺ (stars, inertialInfo)
  let actions =  calculateActions desiredAttitude attitude
  returnA ─≺ actions
```

The auxiliary functions *calculateAttitude′* and *calculateActions′* are now unnecessary, as their fault tolerance handling is now *embedded in the monad*. This last example demonstrates the elegance of this solution: we have incorporated a notion of uncertainty in our function without making it any more complex than it originally was.

In the following section, we will see how this can be expanded on, by adding more structured information in our uncertainty factors.

4 Fault tolerance and voting

Fault tolerance is usually implemented by introducing *redundancy*, that is, multiple units that perform the same or similar actions. Redundancy can be used either to detect failures (by comparing the results of different redundant units), to mask failures (by averaging the results of different redundant units), or to recover from failure (by discarding faulty units and replacing them by their redundant counterparts) (Butler, 2008).

In the previous section, we saw how to calculate using values from multiple, potentially unreliable sensors. Because certainty is always between 0 and 1 and we use the product as the monoidal operation, it can remain stable and decrease, but never increase. Fault tolerance methods, however, are expected to increase the reliability of a system.

We can address this limitation by adding new functions that manipulate signals from different sensors and produce a result with higher certainty.

As an example, let us consider the possibility (not necessarily realistically) that a satellite has three star trackers and tries to use the extra information to calculate an average position. We obtain data from all sensors, together with tolerance factors, and determine that this approach is reliable so long as only one star tracker fails. We use $\lambda x \rightarrow 1 - x$ to invert reliability factors and calculate as follows:

```
starTrackerAvg' :: MSF (FaultToleranceT Certainty m) () StarInfo
                → MSF (FaultToleranceT Certainty m) () StarInfo
                → MSF (FaultToleranceT Certainty m) () StarInfo
                → MSF (FaultToleranceT Certainty m) () StarInfo
starTrackerAvg' starInfo1 starInfo2 starInfo3
   = proc () → do
       (f1, d1) ← runFaultToleranceS starInfo1 ─≺ ()
```

```
    (f2, d2) ← runFaultToleranceS starInfo2 ⤙ ()
    (f3, d3) ← runFaultToleranceS starInfo3 ⤙ ()
      -- A receiver fails if it does not work
    let fails1   = 1 − f1
        fails2   = 1 − f2
        fails3   = 1 − f3
      -- Two fail if any combination of two receivers fail
    let twoFail = fails1 ∗ fails2 + fails2 ∗ fails3 + fails1 ∗ fails3
      -- At least two work if two don't fail
    let twoWork = 1 − twoFail
    arrM (uncurry liftUnreliableT) ⤙ (twoWork, starTrackerAvg d1 d2 d3)
starTrackerAvg :: StarInfo → StarInfo → StarInfo → StarInfo
starTrackerAvg = ...
```

Note that, in this case, we enclose the star tracker polling operations by the function *starTrackerAvg′*, using an *MSF* that gets the data from each sensor and averages it. This is because we want the reliability factors of each star tracker data not to influence our result directly, but only via the fault tolerance method exemplified above. Conceptually, this denotes that they are part of different FCRs.

Our fault-tolerant control step MSF is now slightly different, as it needs to pass three star trackers as inputs for this function:

```
controlSystemStep′ ::  Attitude
                     → MSF (FaultToleranceT Certainty m) Attitude [Action]
controlSystemStep′ initialAttitude = proc (desiredAttitude) → do
  stars       ←  starTrackerAvg′ starTrackerInfo1′
                                 starTrackerInfo2′
                                 starTrackerInfo3′ ⤙ ()
  inertialInfo ← imuSense′ ⤙ ()
  attitude    ←  feedback initialAttitude (arr (dup ∘ calculateAttitude))
                    ⤙ (stars, inertialInfo)
  let actions  =  calculateActions desiredAttitude attitude
  returnA ⤙ actions
```

Simulating this new experiment shows that our confidence in the result has increased due to the more reliable fault tolerance mechanism:

```
ghci > embed (runFaultToleranceS (controlSystemStep initialAttitude))
             [destAttitude, destAttitude, destAttitude]
[(0.9603, [...]), (0.9603, [...]), (0.9603, [...])]
```

This approach could be generalized to perform other kinds of recovery mechanisms. For example, we could make *starTrackerAvg* operate over lists and ignore trackers whose current values are too far from the average, increasing the overall reliability factors.

5 Probability

Let us now enrich the type used to represent reliability with further structure, indicating possible values and their probabilities. We resort to an existing representation for probabilities (Erwig & Kollmansberger, 2006), which we briefly introduce below. We later explore how to combine this representation with MSFs to obtain reactive specifications with probability distributions.

5.1 Probabilities in Haskell

The representation of a type for probabilities has been the subject of prior study in Haskell. It is not our goal to define a custom type for probabilities but, rather, to show that, with few properties, we can combine existing, orthogonal representations of probability distributions with MSFs. In this paper, we follow the description in Erwig & Kollmansberger (2006) due to conciseness and elegance. We only require that we can construct a monad and leave aside the discussion of which specific representation to use.

5.1.1 Probability distributions

We first introduce an abstract type constructor *T* that represents a probability distribution. The details of *T*'s internal definition are purposefully hidden:

```
newtype T prob a
```

For example, the type *T Double Bool* represents the distribution of an event being *True* or *False* that expressed as probabilities with type *Double*. This flexibility lets us use alternative types with different precision to represent probabilities.

For clarity, we define

```
type Dist        = T Probability
type Probability = Rational
```

We can represent values with different probability distributions using multiple auxiliary functions. For example, we can represent the possible results of throwing a perfect die, with uniform distribution, as follows:

```
die :: Dist Int
die = uniform [1 . . 6]
```

This value can be examined directly in a session[3], showing the spread of the distribution with the probability of each individual event:

```
ghci > die
fromFreqs [(1, 0.16), (2, 0.16), (3, 0.16), (4, 0.16), (5, 0.16), (6, 0.16)]
```

We can also query the probability of individual events, for instance, to obtain the probability of obtaining more than 4 if we throw a die:

```
ghci > (>4) ?? die
0.33
```

[3] The numbers in this GHCi session are cropped to two decimals for readability.

We can also create normal distributions, or distributions from specific lists of events and frequencies:

$$\begin{aligned}&\mathit{height} :: \mathit{Dist}\ \mathit{Double}\\&\mathit{height} = \mathit{normal}\ [55, 60, 65, 70, 75, 80, 85]\\&\mathit{probSuccess} :: \mathit{Dist}\ \mathit{Bool}\\&\mathit{probSuccess} = \mathit{fromFreqs}\ [(\mathit{True}, 0.55), (\mathit{False}, 0.45)]\end{aligned}$$

Many other auxiliary functions, types, and distributions are available in this library. A crucial observation is that *T prob* is a monad if *prob* is a number, which means that it can be used in place of *FaultTolerance* to capture different possible results of reactive components and their probabilities. It is worth noting, though, that the interpretation of probabilities from monadic calculations makes the different distributions potentially dependent, as the type signature of the bind operator, specialized for this monad, is:

$$(\gg\!=) :: \mathit{Dist}\ a \to (a \to \mathit{Dist}\ b) \to \mathit{Dist}\ b$$

that is, the calculation of the probability distribution for the second argument may depend on a specific occurrence of the first.

As described in Erwig & Kollmansberger (2006), there is a function in the probability library, *joinWith*, that allows us to merge two distributions that are truly independent from one another. We can obtain the same result using $(\gg\!=)$ with a function that ignores its argument as the second argument or by using $(\gg)$.

5.2 Probabilities in reactive systems

Since *T a* is a *Monad* if *a* is a number, we can immediately use it together with MSFs. We can use probabilities with MSFs to actually modify or alter the value, modeling the introduction of noise in signals. For example, we can introduce values slightly different from that of an ideal star tracker (assuming we can multiply it). We indicate that the probability of those differences is 0.1%, while we have a probability of 99.6% of obtaining the correct value:

$$\begin{aligned}&\mathit{starTrackerSense}' :: \mathit{MSF}\ \mathit{Dist}\ ()\ [\mathit{StarInfo}]\\&\mathit{starTrackerSense}' = \mathit{starTrackerSense} \ggg \mathit{arrM}\\&\quad (\lambda s \to \mathbf{let}\ [s1, s2, s3, s4] = \mathit{map}\ (\mathit{adjustStarInfo}\ s)\ [0.90, 0.95, 1.05, 1.10]\\&\qquad\qquad \mathbf{in}\ \mathit{fromFreqs}\ [(s1, 0.001), (s2, 0.001), (s3, 0.001), (s4, 0.001), (s, 0.996)])\\&\mathit{adjustStarInfo} :: \mathit{Num}\ a \Rightarrow [\mathit{StarInfo}] \to a \to [\mathit{StarInfo}]\\&\mathit{adjustStarInfo} = \ldots \quad \text{-- Defined elsewhere}\end{aligned}$$

Just like before, adapting all operations makes our types change, but our main MSF remains unchanged:

$$\begin{aligned}&\mathit{controlSystemStep}' :: \mathit{Attitude} \to \mathit{MSF}\ \mathit{Dist}\ \mathit{Attitude}\ [\mathit{Action}]\\&\mathit{controlSystemStep}'\ \mathit{initialAttitude} = \mathbf{proc}\ (\mathit{desiredAttitude}) \to \mathbf{do}\\&\quad \mathit{stars} \leftarrow \mathit{starTrackerSense}' \prec ()\\&\quad \mathit{inertialInfo} \leftarrow \mathit{imuSense}' \prec ()\\&\quad \mathit{attitude} \leftarrow \mathit{feedback}\ \mathit{initialAttitude}\ (\mathit{arr}\ (\mathit{dup} \circ \mathit{calculateAttitude}))\end{aligned}$$

```
                  ⤙ (stars, inertialInfo)
    let actions   = calculateActions desiredAttitude attitude
    returnA ⤙ actions
```

To explore values inside *Dist*, we define an auxiliary function, *runDistS*, that extracts the values of an MSF in the *Dist* monad, putting the probabilities in the output[4].

If we try this in a session, we get a probability distribution of all possible outputs (formatted for readability):

```
ghci > embed (runDistS $ controlSystemStep' initialAttitude) [destAttitude]
[([Action...], 0.001), ([Action...], 0.001), ([Action...], 0.001)
, ([Action...], 0.001), ([Action...], 0.996)
]
```

Another possible use of probabilities is to use them to indicate which kinds of faults are taking place. This will be discussed later in Section 6.

5.3 Fault tolerance with probabilities

Due to the way that probabilities are represented here, adding fault tolerance in this case is easier than with other constructs. The definitions of *bind* and *return* for the *Dist* monad already calculate the new probabilities of the different possible combinations of inputs and outputs, which makes it easier to write expressions that combine probabilities. In this case, manually calculating the inverses is not necessary, and an occurrence of the same event multiple times will be combined to provide an accumulated event with the addition of both probabilities.

```
starTrackerAvg' :: MSF Dist () StarInfo
                → MSF Dist () StarInfo
                → MSF Dist () StarInfo
                → MSF Dist () StarInfo
starTrackerAvg' starInfo1 starInfo2 starInfo3
  = proc () → do
      d1 ← starInfo1 ⤙ ()
      d2 ← starInfo2 ⤙ ()
      d3 ← starInfo3 ⤙ ()
      returnA ⤙ starTrackerAvg d1 d2 d3
starTrackerAvg :: StarInfo → StarInfo → StarInfo → StarInfo
starTrackerAvg = ...
```

If we now explore the probabilities with averaging, we see that it is more likely to be wrong: it will only be perfectly correct if all star trackers work perfectly. However, due to the average eliminating part of the error, even when it is incorrect, it would normally be wrong by a smaller amount.

[4] Internally, this particular monad is defined similarly to the so-called "broken" list transformer *ListT*. Since MSFs already support working with *ListT*, defining this function is trivial and we obviate that detail.

```
ghci > embed (runDistS $ controlSystemStep′ initialAttitude) [destAttitude]
[(Action..., 0.988047936) , (Action..., 2.985015e−3), (Action..., 2.982025e−3)
, (Action..., 2.979036e−3), (Action..., 2.976048e−3), (Action..., 1.1958e−5)
 ...
]
```

It is worth noting that the above definition suggests parallelism and independence between the star trackers, but the expression is evaluated by first evaluating *starInfo1*, passing the output to a calculation that then evaluates *starInfo2*, and passing that result to a calculation that evaluates *starInfo3*. These three MSFs are combined by means of the monadic bind (≫=) operator, in principle enabling the individual probabilities of the results of each calculation to be related to one another (i.e., not independent). However, in this specific example, the star trackers ignore that input and produce a probability distribution that does not take the previous calculations into account, suggesting that the probability of faults or errors in their outputs are independent.

To capture the intention of true parallelism of independent *MSF*s, we could define a specialized combinator that broadcasts the inputs to several *MSF*s and collapses the monads of the outputs, in a parallel and independent fashion. For example, for two MSFs, we could define:

$$(|||) :: \mathit{MSF}\ \mathit{Dist}\ a\ b \rightarrow \mathit{MSF}\ \mathit{Dist}\ a\ c \rightarrow \mathit{MSF}\ \mathit{Dist}\ a\ (b, c)$$

implemented internally using *joinWith* (,), described earlier in this section, to join the results of independent events. The new implementation would then be:

```
starTrackerAvg′ starInfo1 starInfo2 starInfo3
   = proc () → do
      ((d1, d2), d3) ← starInfo1
                      ||| starInfo2
                      ||| starInfo3 −≺ ()
      returnA −≺ starTrackerAvg d1 d2 d3
```

6 Fault sets and fault analysis

The previous approach provides an understanding of the reliability of a system, but it does not help determine *what caused the failure*. Let us now use the same abstraction to indicate the possible faults that may have affected the calculations.

In particular, we want to focus on the case in which we know a fault may take place, but not whether it actually has, known as a *transmissive* fault. For example, a bit flip in the memory of the star tracker may corrupt the results, and, if we are building a data transmission system for a satellite, we may not have sufficient application knowledge to understand which values may be completely wrong.

Even if we do not handle a particular kind of fault at a level, we might want to have a clear, machine-checked specification of the faults that we consider and those we do not. Let us first define a type that contemplates all possible faults in our *fault model*:

```
data Fault = StarTrackerBatteryError
           | StarTrackerFixNotFound
           | StarTrackerProcessingError
           | ...
  deriving (Ord, Eq)
```

We now use sets with union and the empty set as the monoidal operations for the fault tolerance monad, so our *starTrackerSense′* function becomes

```
starTrackerSense′ :: MSF (FaultToleranceT (Set Fault) m) () StarInfo
starTrackerSense′ = starTrackerSense >>> arrM
                         (liftUnreliableT
                            [StarTrackerBatteryError
                            , StarTrackerFixNotFound
                            , StarTrackerProcessingError
                            ]
                         )
```

We assume that *imuSense′* changes accordingly. Like in previous occasions, our main function changes its type, but the definition remains the same. This demonstrates a key strength of MSFs to specify and extend reactive systems with additional features:

```
controlSystemStep′ :: Attitude → MSF (FaultToleranceT (Set Fault) m) Attitude [Action]
controlSystemStep′ initialAttitude = proc (desiredAttitude) → do
  stars       ← starTrackerSense′ −≺ ()
  inertialInfo ← imuSense′        −≺ ()
  attitude    ← feedback initialAttitude (arr (dup ∘ calculateAttitude))
                   −≺ (stars, inertialInfo)
  let actions  = calculateActions desiredAttitude attitude
  returnA −≺ actions
```

We run one step of this *MSF* to obtain a list of all possible faults that may have affected the result. Due to how the monad affects all *MSF*s connected to the *starTrackerSense′*, the top level *MSF* includes all the faults that may have affected the star tracker:

```
ghci > embed (runFaultToleranceS $ controlSystemStep′ initialAttitude) [destAttitude]
[(Set [StarTrackerBatteryError, StarTrackerFixNotFound, StarTrackerProcessingError],
  , (Action...))]
```

In Section 4, we introduced voting to eliminate some classes of faults. Let us adapt that example to this new case, to show the benefits of adding explicit faults. Assuming that this fault tolerance mechanism only deals with processing errors in the star trackers, but not with other kinds of faults (e.g., battery errors), we define

```
starTrackerAvg′ :: MSF (FaultToleranceT (Set Fault) m) () StarInfo
                → MSF (FaultToleranceT (Set Fault) m) () StarInfo
                → MSF (FaultToleranceT (Set Fault) m) () StarInfo
                → MSF (FaultToleranceT (Set Fault) m) () StarInfo
```

```
starTrackerAvg′ starInfo1 starInfo2 starInfo3
  = proc () → do
      (f1, d1) ← runFaultToleranceS starInfo1 ─≺ ()
      (f2, d2) ← runFaultToleranceS starInfo2 ─≺ ()
      (f3, d3) ← runFaultToleranceS starInfo3 ─≺ ()
        -- A receiver fails if it does not work
      let handled = [StarTrackerProcessingError]
      let faults1 = f1 \ handled
          faults2 = f2 \ handled
          faults3 = f3 \ handled
      let faults = faults1 ‘union‘ faults2 ‘union‘ faults3
      arrM (uncurry liftUnreliableT) ─≺ (faults, starTrackerAvg d1 d2 d3)
starTrackerAvg :: StarInfo → StarInfo → StarInfo → StarInfo
starTrackerAvg = ...
```

Running an adapted version of *controlSystemStep′* with this definition would no longer include *StarTrackerProcessingError* in the output, but would still include other kinds of faults. To make sure we do not remove faults that we do not handle, we explicitly list the faults that this method may deal with and put them in the output.

Because *FaultToleranceT* is a monad transformer, we can combine this method with previous ones. For instance, we could state that a value may fail, or, separately, that it may be slightly inaccurate, using the monad *FaultToleranceT* (*Set Fault*) *Dist*. We could also use a multiset or a more complex structure to include multiple possible errors of the same kind and state that the fault tolerance mechanisms can help recover from up to a number of possible simultaneous faults, but not more. We have decided to include a fixed fault set but, as demonstrated in Section 3, different faults can be included in the fault set dynamically during execution.

7 Fault injection and randomized testing

In previous sections, we saw how to annotate reactive networks with different kinds of reliability information and how fault-tolerant mechanisms needed to manipulate that information so that the conclusions that we drew from the system were correct. Normally, the evaluation of such mechanisms was done manually, based on one or very few samples.

A thorough, systematic evaluation of fault tolerance requires several steps. First, we must provide a clear description of the reliability or fault information that we wish to capture about the system. In the case of fault sets and fault models, this implies capturing what faults are representable in the system and how they manifest. This process depends on how we model the system itself: some faults may not be distinguishable from others in our specific model. Second, to be able to evaluate the fault tolerance mechanisms introduced, we must have a way to capture system requirements and test whether a particular implementation fulfills them. Third, we must be able to evaluate the system in the presence of faults, by injecting faults during execution or simulation. Finally, we must have a way of comparing the behavior of the system with no faults present, when faults are injected, and when fault tolerance is introduced, to determine if the fault tolerance mechanisms are sufficient for the conditions expected in the real world.

In this section, we describe how these steps can be achieved in *MSF*s, using formalizations based on temporal logic to capture system properties, using MSFs that inject faults with different probabilities or that alter the system in a controlled manner, and using QuickCheck to randomly explore the input space and evaluate the results.

7.1 Injecting faults

In prior sections, we saw how different kinds of fault information can be incorporated in reactive systems. A thorough exploration of faults in a reactive system requires that we can inject the different kinds of faults that we consider, in the locations where those faults can occur. It is therefore a problem of determining how faults affect values and which values they affect.

Simulating faults. How faults affect values is specific to the fault model we are considering. In prior examples, we described a fault model that proposed a class of faults known as *malign* faults, those in which a value changes in a way that is not trivially detectable. To evaluate the control systems we defined before, we could, for example, define a fault injection MSF that changes *StarInfo* affecting it slightly: enough to affect the mission, but not enough that the value is clearly out of range. Assuming that we can multiply it by a scalar with the function $(*\hat{})$, and that a variation of 5% is enough to make the value incorrect, we could implement such a fault injection MSF by the following expression:

$$\begin{aligned} &\mathit{transmissiveFault} :: \mathit{MSF}\ (\mathit{FaultToleranceT}\ (\mathit{Set\ Fault})\ m)\ \mathit{StarInfo}\ \mathit{StarInfo} \\ &\mathit{transmissiveFault} = \mathit{arr}\ (*\hat{}\ 0.95) \\ &\qquad\qquad \ggg \mathit{arrM}\ (\mathit{liftUnreliableT}\ [\mathit{StarTrackerProcessingError}]) \end{aligned}$$

We can modify a larger *MSF* by introducing this kind of fault in different places and checking whether the fault affects system operations. For example, we can modify the fault-tolerant star tracker definition on page 19, so that it uses the following MSF for the first star tracker:

$$\begin{aligned} &\mathit{starInfo1}' :: \mathit{MSF}\ (\mathit{FaultToleranceT}\ (\mathit{Set\ Fault})\ m)\ ()\ \mathit{StarInfo} \\ &\qquad\qquad \to \mathit{MSF}\ (\mathit{FaultToleranceT}\ (\mathit{Set\ Fault})\ m)\ ()\ \mathit{StarInfo} \\ &\mathit{starInfo1}' = \mathit{transmissiveFault} \lll \mathit{starInfo1} \end{aligned}$$

Of course, in a more realistic scenario, we might want to add faults randomly, only with certain probability. For example, we could modify *transmissiveFault* to inject a fault only 50% of the time or to inject a fault whose magnitude follows some random distribution. Although we could include the probability calculation as part of the MSF itself, it is easiest to externalize it and rely on existing mechanisms to generate those probabilities, like the random testing framework QuickCheck. In the following example, we use an ad hoc type to represent fault injection factors, simply for readability:

$$\mathbf{data}\ \mathit{FaultInjection} = \mathit{FaultInjection}\ \{\mathit{faultInjectionFactor} :: \mathit{Double}\}$$

We now make this factor available via a *Reader* monad and use it to transform the values by affecting them by a given factor. Like before, we state that the fault information is also included in the fault set:

```
tranmissiveFault :: MSF (FaultToleranceT (Set Fault) (ReaderT FaultInjection m))
                        Double
                        Double
tranmissiveFault = valueTransformation >>> faultSetTransformation
  where
    valueTransformation    =
      liftTransS (arrM (λi → (∗i) <$> faultInjectionFactor <$> get))
    faultSetTransformation = arrM (liftUnreliableT [TransmissiveFault])
```

The above definition makes use of *liftTransS*, which lifts an MSF operating on a monad (in this case, *ReaderT FaultInjection m*) into one operating on a transformer applied to that monad (*FaultToleranceT* (*Set Fault*) (...)).

The difference between this MSF and the previous one is that this MSF exposes a fault injection interface all the way to the top level. We could use a session to evaluate a reactive system with fault injection with a particular *faultInjectionFactor*, by just passing the desired value to *runReader* when evaluating the top-level monadic result. We could also use a system with QuickCheck to randomly generate fault injection factors and evaluate a property of the system.

In principle, the exact same mechanism would work to introduce a benign fault, with the difference that a benign fault would be large enough to be obviously a fault and not a good value. Introducing an omissive fault would require a notion of asynchronicity, which can be incorporated into MSFs in multiple ways, the simplest of which would be make outputs optional with *Maybe*. Different ways of introducing asynchronicity in MSFs have been studied in Perez (2017) and Bärenz & Perez (2018).

Modeling system properties. Evaluating the fault tolerance mechanisms in MSFs requires specifying the expected observable behavior and comparing it with the actual system during simulation. In the case of the prior example *starTrackerAvg′*, we could define correct behavior as the result being close enough to the correct value in the presence of faults.

Like in other kinds of systems, it is possible to insert assertions in MSFs that detect property violations during simulation. Because MSFs can have additional side effects, we can use a referentially transparent method to add ad hoc assertions while only affecting the types of MSF. For example, if assuming that *starInfo2* works correctly, we could define an MSF that verifies that the average calculated by *starTrackerAvg′* is close to the perfect value provided by *starInfo2*:

```
ftAssertion
  :: MSF (FaultToleranceT (Set Fault) (ReaderT FaultInjection m)) () StarInfo
  → MSF (FaultToleranceT (Set Fault) (ReaderT FaultInjection m)) () StarInfo
  → MSF (FaultToleranceT (Set Fault) (ReaderT FaultInjection m)) () StarInfo
  → MSF (WriterT String (FaultToleranceT (Set Fault) (ReaderT FaultInjection m)))
```

```
        () StarInfo
  ftAssertion starInfo1 starInfo2 starInfo3 =
    (avg &&& ref) ⋙ (identity &&& property) ⋙ arr fst
  where
    avg = liftTransS (starTrackerAvg′ starInfo1 starInfo2 starInfo3) ⤙ ()
    ref  = liftTransS (starInfo2) ⤙ ()
    property = arr closeEnough ⋙ arrM (λt → unless t $ put "Error")
    closeEnough ((_, dAvg), (_, dRef)) = abs (dAvg − dRef) < threshold
```

The above definition makes use of *liftTransS*, which lifts an MSF operating on a monad (in this case, *FaultToleranceT* (*Set Fault*) *m*)) to one operating on a transformer applied to that monad (*WriterT String*...). The use of the *FaultInjection* type allows us to provide information during simulation (e.g., when calling this MSF with embed) that becomes available to the internal star tracker with the transmissive fault, without having to be manually routed down in these signals.

Nevertheless, as the complexity of systems grows, writing and including these assertions can be cumbersome. Ideally, we would like to use a high-level language to specify system properties, such as temporal logic, which can be implemented on top of MSFs (Perez & Nilsson, 2018), both using past-time temporal logic in the form of assertions within our network, and using bounded future time linear temporal logic from outside the network, seeing the complete MSF network as a black box.

An MSF equivalent to the previous example could have been defined as the following LTL specification, with *avg*, *ref* and *closeEnough* defined as before:

```
ltlProperty = Always (Prop ((avg &&& ref) ⋙ arr closeEnough))
```

The evaluation of this temporal property can be done using QuickCheck, by generating multiple input streams and other additional data, with a function *evalMSF* provided by our library.

The use of QuickCheck together with temporal logic and fault injection can help answer multiple questions, like *does this property hold if the probability of some kind of fault is X* (where X can be a specific probability, a range, or a distribution).

7.2 Modeling satellite networks

The introduction of faults in *MSF* architectures can be done in a systematic way, following the definition of a fault model that classifies the kinds of faults that the system must tolerate, as described in Perez *et al.* (2019). For example, in a satellite swarm that performs multiple observations of the same event from different locations to provide redundant measurements, we may be interested in exchanging information to achieve consensus. In the presence of asynchronicity, transmissive, and omissive faults, consensus may not be guaranteed.

System model. A way to structure a model of such network is to see satellites as MSFs, each reading incoming data from sensors and the network, and producing both new

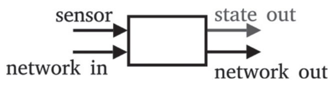

Fig. 2. Model of a satellite as an MSF.

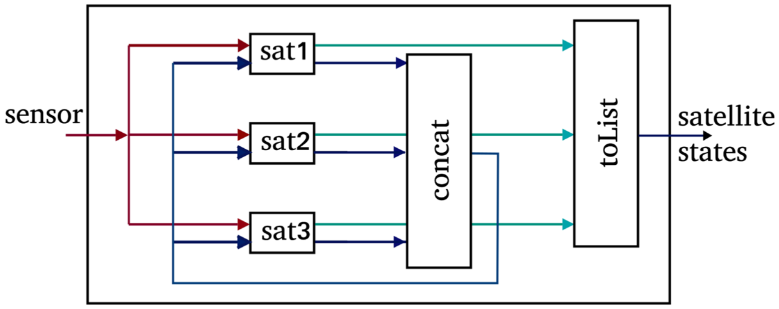

Fig. 3. Model of a satellite swarm as an MSF network.

messages and a view of their internal state (for debugging purposes) (Figure 2). For example, if we assume that the satellites are measuring some numeric value, the value obtained from sensors could be defined as:

$$\textbf{type}\ \mathit{SensorData} = \mathit{Double}$$

However, to implement distributed fault tolerance, each satellite needs to know which values other satellites have obtained so far, which we could capture with the following type, where the first *Int* represents the id of the satellite:

$$\textbf{type}\ \mathit{SatelliteState} = [(\mathit{Int}, \mathit{Maybe}\ \mathit{SensorData})]$$

Satellites can send each other their readings in network messages which are broadcast to the whole network:

$$\textbf{data}\ \mathit{NetworkMsg} = (\mathit{Int}, \mathit{SensorData})$$

A satellite processing MSF would then have the following type, as depicted in Figure 2:

$$\begin{aligned}\textbf{type}\ \mathit{SatelliteMSF}\ m = \mathit{MSF}\ m\ &(\mathit{SensorData}, [\mathit{NetworkMsg}])\\ &(\mathit{SatelliteState}, [\mathit{NetworkMsg}])\end{aligned}$$

With minor additions, we could connect several such MSFs and broadcast outgoing messages to all other MSFs (satellites) (Figure 3).

Fault model and fault injection. A suitable fault model to explore fault injection in this architecture is to see faults as benign (trivially wrong) or malign. The latter group can be further subdivided into omissive faults (messages dropped) and transmissive faults (messages altered). Just like in previous examples, we can define MSFs that inject different kinds of faults, depending on some probability factor P available via a Reader monad (Figure 4).

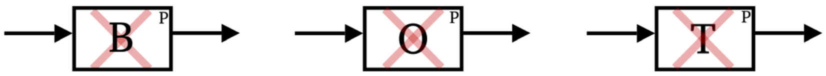

Fig. 4. Fault injection MSFs for benign, omissive, and transmissive faults.

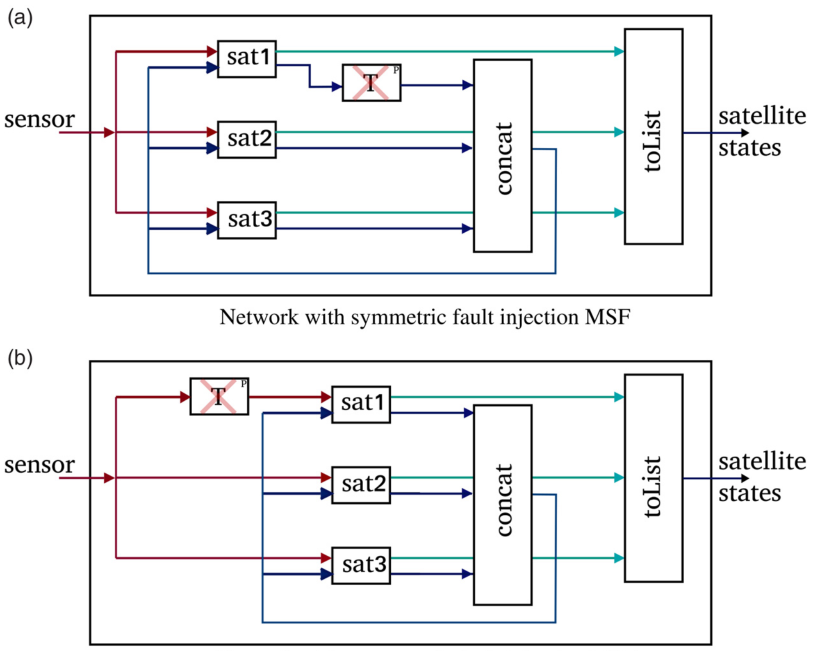

Fig. 5. Networks with symmetric (a) and asymmetric (b) fault injection.

Faults can be made symmetric or asymmetric by inserting them in different parts of the network (Figure 5). Inserting a fault in a message before the message is broadcast to other nodes will make it symmetric (all nodes will receive the altered message). On the other hand, injecting a fault in a message after broadcast will only affect a specific node, making the fault asymmetric.

Requirements and system properties. The use of redundancy can help satellites detect and mask some of those faults. Properties of distributed consensus can be modeled as temporal properties, following the same approach presented in this section, and verified using QuickCheck. Assuming, for the sake of the example, a discrete notion of time, we define consistency as a combination of two properties (Kshemkalyani & Singhal, 2008):

- Validity: If a satellite s observes a value v at a time t, other satellites will associate value v to satellite s at time $t + 1$.
- Agreement: any two satellites agree on the last known value for a third satellite.

For example, the first property of validity can be captured in past-time temporal logic as $\forall i, j.(\bigcirc(\Box(i \neq j \Rightarrow same\_as\_last(i, j, i))))$ where we define $same\_as\_last(i, j, v)$ as the temporal expression $values_j(t-1)[v] = values_i(t)[v]$, where t represents the current time and $values_j(t)[v]$ represents the state of satellite v at time t as captured by satellite j, observable as the global output of the network (satellites states in Figure 3). We can evaluate this property with QuickCheck by exploring the space of possible satellites (having QuickCheck randomly generate pairs (i, j) for input streams of different lengths, as well as different probability factors for different kinds of faults.

7.3 Modeling faults in the Space Shuttle

Let us now model a small part of NASA's Space Shuttle and demonstrate a fault that occurred during the STS-124 mission. This is merely a schematic example to illustrate the ideas in this paper, based on publicly available documents, but it is not based on the actual system design. For further details, including technical details on the fault that we model and the components affected, see Driscoll (2008).

In May 2008, a NASA's Space Shuttle was loading fuel when a fault occurred in the system: a diode cracked and started behaving like a capacitor. This diode was part of a box that sends messages, via a multi-drop bus, to four redundant computers. Due to the way that the diode was behaving, different computers were getting different messages: a perfect example of what is known as a Byzantine fault (Lamport *et al.*, 1982).

System model. The four redundant computers were connected to a system that determined the correct value using fault tolerance. We can model this example with the following MSF, which specifies that the input is sent to all four buses, and their values are made available to a fault tolerance subsystem:

```
subsystem :: MSF m Input Output
subsystem = proc i → do
  b1 ← bus1 −≺ i
  b2 ← bus2 −≺ i
  b3 ← bus3 −≺ i
  b4 ← bus4 −≺ i
  faultTolSys −≺ (b1, b2, b3, b4)
```

To prevent faulty computers from affecting the system, the fault tolerance mechanism would disable or discard values from a computer that had been flagged as faulty.

A schematic model this fault-tolerant system follows. We use the auxiliary functions *ftMask* and *valid*, which calculate, respectively, the result from a number of valid general purpose computers (GPCs), and whether a value provided by one GPC is close enough to the ideal to be considered valid.

```
faultTolSys :: MSF Fault Input Output
faultTolSys = feedback [True, True, True, True]
                  $ proc ((i1, i2, i3, i4), cs@[c1, c2, c3, c4]) → do
   (b1, f1) ← runFaultToleranceS gpc1 −≺ i1
   (b2, f2) ← runFaultToleranceS gpc2 −≺ i2
   (b3, f3) ← runFaultToleranceS gpc3 −≺ i3
```

```
(b4,f4) ← runFaultToleranceS gpc4 ⤙ i4
let bs = [b1, b2, b3, b4]
  -- Use only values from “valid” computers
let bs′ = [b | (b, c) ← zip bs cs, c]
  -- Calculate most frequent value
let bRef = ftMax bs
  cs′ = zipWith (λb c → c ∧ (valid b bRef)) bs cs
  -- A receiver fails if it does not work
let handled = [GPCError]
let faults1  = f1 \ handled
    faults2  = f2 \ handled
    faults3  = f3 \ handled
    faults4  = f4 \ handled
let faults = faults1 ‘union‘ faults2 ‘union‘ faults3 ‘union‘ faults4
arrM (uncurry liftUnreliableT) ⤙ (faults, (bRef, cs′))
```

Injecting faults. While this mechanism handles up to two simultaneous faults in computers, it does not handle faults in a diode at all. We could modify the definition *subsystem* to inject transmissive faults in the buses, by exposing a record with four-fault injection or probability factors via a Reader monad. By making these factors available to the top level, we could control not only when a fault is injected, but also that it is not injected in more than one processor at a time, making it transmissive. Notice that how the signal affects values can make it benign or malign in our fault classification. Whether all values are affected by the same amount or not, which is controlled using different fault injection factors, determines whether the fault is symmetric or asymmetric. By placing one-fault injection MSF in each bus, we enable the possibility of asymmetry, just like in the prior example, by placing the fault injection MSFs immediately before each satellite, and after the broadcast, make the fault asymmetric.

System requirements and temporal properties. When a fault affects a CPU in a way that the fault tolerance system can detect it, it assumes the CPU is faulty and that it must be disabled, assuming that the likelihood that the bus or a component before the bus be faulty is lower than the CPU being faulty. We can check this example using a reference *MSF* as “good” value for testing purposes and evaluating the temporal property:

$$ltlProperty = Always\ (Prop\ ((subssytem\ \&\&\&\ ref) \ggg arr\ (uncurry\ valid)))$$

This property fails with a limited number of tests. When two computers have been discarded, then the system can no longer differentiate between a fault in a third computer and the correct value, which is exactly what happened during the Space Shuttle mission. In the next section, we will see how we could have obtained a hint during compilation that this failure could have taken place in our system, by encoding faults as values at type level, which would have made the fault-tolerant four-way computer not exhibit a single computer fault, but the *subsystem* would have included faults in the diode or in the communication buses.

7.4 Discussion

This section provides tools for more systematic evaluation of fault tolerance, but its reach is still limited. For example, the creation of an appropriate fault model is crucial for a proper evaluation, and it requires expert knowledge in the specific domain. A fault model is never absolutely complete, and faults may easily be incompletely or incorrectly categorized. Some hazards are not considered probable enough to be included in the fault model, although new evidence from experiments may require that they be introduced in future models. The independence between faults in different parts of the system is only guaranteed for components in separate FCRs, and with respect to specific faults. For another example, determining where and how fault tolerance mechanisms can be introduced is a task that affects not only the behavior of the system, but its weight and cost, and cannot be done in isolation without requiring a re-evaluation of the mission, potentially affecting the original requirements.

Generally, at the level of abstraction provided by MSFs, we can always inject faults in every signal, and we can inject faults that modify values, that drop values, that duplicate values, and faults that delay values.

Both modeling and injecting these kinds of faults is currently a manual process. There are two limitations in the current implementation that prevent us from performing fault injection automatically. The first has to do with characterizing how faults affect values, in particular due to the full polymorphism of the primitives and combinators in the *Arrow* class. Automatically injecting, for example, a transmissive fault would require being able to alter any value of any type, for which we currently have no mechanism in Haskell. We could limit MSFs to operate on values we can alter (or serialize/deserialize), possibly using a type class to capture that constraint. This, however, would limit our language to operate only on values that instantiate that class, which is currently not allowed by the *Arrow* class or by the notation overloading mechanisms available in Glasgow Haskell Compiler (GHC). A second limitation stems from the fact that, to inject a fault in a network, we must be able to determine where in the network it is injected, for which we need an understanding of the structure of the reactive system. Several alternatives exist the most promising being the use of a deeply embedded Domain Specific Language (DSL), which would allow us to inspect it and modify it to inject faults, and to compile MSFs into, for example, C code that could run on an embedded system.

8 Parameterized MSFs

The approach presented in previous sections allows us to conclude an execution with a set of faults. While useful, that does not provide any static guarantees about the presence or absence of certain faults: such information is only obtained during execution.

An alternative approach would be to encode specific faults as values at type level. We can take advantage of dependently typed programming and encode the possible faults at a particular point using a *Set* of *Fault*s. In Haskell, existing formalizations of type-level sets[5] make this possible (Orchard & Petricek, 2015).

[5] https://hackage.haskell.org/package/type-level-sets

For example, in our case, we might have wanted to type *starTrackerSense′* as follows:

```
starTrackerSense′ :: MSF
                       (FaultToleranceT
                         (Set′[StarTrackerBatteryError, StarTrackerFixNotFound
                              , StarTrackerProcessingError])
                         m)
                       () StarInfo
```

While this is possible by means of the aforementioned libraries and with multiple type-dependent extensions in GHC, a crucial observation is that the monad changes. In order to make this work for MSFs, we need to extend them to support indexed or parametric monads. In the rest of this section, we extend MSFs accordingly and we return to this point in the next section to show the definition of this function with type-level fault information.

Let us introduce a new interface to work with MSFs that extend the functions presented before. For the purposes of understanding, we also introduce a **newtype** with the definition of MSF, which we use to give model definitions of the new operations available. In the rest of the discourse, we assume that this new interface substitutes the previous one.

8.1 Parametric monads

If we specify the set of possible faults for an operation using a closed set, then the *Monad* will change if we specify a different set of possible faults. To work around this limitation[6], we opt for *parameterized monads*, which relax the definition of bind so that the monad can change. The definition of the type class for parameterized monads is divided into two type classes:

```
class Return m where
  returnM :: a → m a
class Bind m1 m2 m3 | m1 m2 → m3 where
  bindM :: m1 a → (a → m2 b) → m3 b
```

Multiple authors have explored different monadic notions with different levels of parametrization, different kinds of constraints, and different categorical meaning. Our goal is to show that, if the monad is allowed to change, we can easily work with type-level sets of faults, which addresses many of our concerns. In practice, some of these proposals are too general, and type systems have trouble giving type signatures to intermediate constructs. We have experimented with different approaches to encode this level of flexibility and we have had most success with *effect monads*, which add an additional parameter to the monad *m*, instead of allowing bind to alter the monad fully. For the rest of the section, we describe our proposal based on these parameterized monads, also known as *polymonads*. Other proposals are discussed in Section 10.

[6] We could have opted to specify type-level fault sets as constraints (Set *s* contains fault *Y*, as opposed to set is *Set′*[*Y*]. While this would make the use of parametric monads unnecessary, it would also complicate type signatures.

8.2 Parametric MSFs

8.2.1 Basic definitions

Like before, an *MSF* is a type that represents a step in a synchronous causal transformation between changing inputs. We define it fully as:

$$\textbf{newtype}\ MSF\ m\ a\ b = MSF\ \{step :: a \rightarrow m\ (b, MSF\ m\ a\ b)\}$$

The function *step* takes an input sample and returns, in a monadic context, an output and a continuation. In general, we assume that *m* is a parameterized *Monad*. While the above type exposes the constructor of *MSF*s, in general, we discourage users from using it and provide the following leak-free causal interface to define and combine *MSF*s.

In this section, we redefine MSFs using the same names that we used before for functions and combinators. This lets us use the same notation, including arrow **proc** notation. In practice, support for re-binding arrow combinators in GHC is limited. Our real implementation uses different names. This simplification does not affect the validity of our claims.

8.2.2 Pointwise transformations

*MSF*s can be transformed by applying a pointwise transformation on the input. While normally this would be Kleisli arrow for a given *Monad* instance, we use the type classes *Return* and *Bind* to define this function so that the MSF works for parameterized monads:

$$\begin{aligned}&arrM :: (Return\ m, Bind\ m\ m\ m) \Rightarrow (a \rightarrow m\ b) \rightarrow MSF\ m\ a\ b\\&arrM\ f = MSF\ \$\ \lambda a \rightarrow f\ a \gg\!= \lambda b \rightarrow returnM\ (b, arrM\ f)\end{aligned}$$

Obviously, one can always apply a pure transformation on the input, so we define, for convenience:

$$\begin{aligned}&arr :: (Return\ m, Bind\ m\ m\ m) \Rightarrow (a \rightarrow b) \rightarrow MSF\ m\ a\ b\\&arr\ f = arrM\ (return \circ f)\end{aligned}$$

8.2.3 Composition

We also provide a way to compose *MSF*s. This is one of the main differences with respect to the previous framework, since the monad now changes:

$$\begin{aligned}&(\ggg)\ ::\ (Return\ m1, Return\ m2, Return\ m3, Bind\ m1\ m2\ m3, Bind\ m2\ m2\ m2)\\&\qquad \Rightarrow MSF\ m1\ a\ b\\&\qquad \rightarrow MSF\ m2\ b\ c\\&\qquad \rightarrow MSF\ m3\ a\ c\\&(\ggg)\ (MSF\ msf1)\ (MSF\ msf2) = MSF\ \$\ \lambda a \rightarrow \textbf{do}\\&\quad (r1, msf1') \leftarrow msf1\ a\\&\quad (r2, msf2') \leftarrow msf2\ r1\\&\quad returnM\ (r2, msf1' \ggg msf2')\end{aligned}$$

8.2.4 Widening

Arrows require a way of applying a transformation to only one input in a pair, leaving the other input unchanged. This definition is straightforward and follows the one presented earlier, except that it is specialized for the interface of parameterized monads:

```
first :: (Return m, Bind m m m) ⇒ MSF m a b → MSF m (a, c) (b, c)
first (MSF msf1) = MSF $ λ(a, c) → do
   (b, msf1′) ← msf1 a
   returnM ((b, c), first msf1′)
```

We can define *second* analogously, although it is not primitive and can also be defined in terms of *first* as *second msf* = *arr swap* ⋙ *firstM msf* ⋙ *arr swap*.

8.2.5 Depending on the past

Finally, and just like before, we provide a way of creating a well-formed feedback loop:

```
feedback :: (Return m, Bind m m m) ⇒ c → MSF m (a, c) (b, c) → MSF m a b
feedback c (MSF msf) = MSF $ λa → do
   ((b′, c′), msf′) ← msf (a, c)
   returnM (b′, feedback c′ msf′)
```

We can also delay the input by one sample, which can be trivially defined in terms of well-formed feedback:

```
iPre :: (Return m, Bind m m m) ⇒ a → MSF m a a
iPre a0 = feedback a0 (arr swap)
   where
      swap (x, y) = (y, x)
```

If we make $m1 \equiv m2 \equiv m3$ in the definitions presented in this section, then the *Return* and *Bind* type constraints simply correspond to standard *Monad* type class constraints, and we obtain an interface equivalent to the one provided before for ordinary MSFs.

9 Encoding faults at the type level

Now that we have extended MSFs to work with different monads, let us come back to our original proposal for fault tolerance monads and present how to extend it to operate with type-level fault sets.

Throughout the rest of the section, we simplify some functions, eliminating trivial constraints from the type signatures. This is limited only to equivalences that can be derived from the monoid laws.

First, we define monad instances for *FaultToleranceT* on fault sets, using the definition of sets as a monoid with union and the empty set as the identity:

```
instance Monad m ⇒ Return (FaultToleranceT (Set′[ ]) m) where
   returnM x = return (Empty, x)

instance Monad m ⇒ Bind (FaultToleranceT (Set x) m)
                        (FaultToleranceT (Set y) m)
                        (FaultToleranceT (Set (x ‘union‘ y) m) where
   bindM (set1, v1) f = do (set2, v2) ← f v1
                           return (set1 ‘union‘ set2, v2)
```

Example. We use the same definition of *Fault* as before. Making *Fault*s into type-level *Fault*s requires the use of several dependent-type programming extensions in GHC that promote values to the type level. The use of *Fault* with set requires the definition of an instance of *Cmp*, a type-level comparison function akin to *compare* for instances of *Ord*, that used to eliminate duplicates from the set. For convenience, we also define instances for *Show*, to inspect specifications using a GHCi session.

In order to represent these instances at the type level in sets, we need to create a *proxy* type. This lets us create values of those faults to fill in the type-level set. While we could opt to use a generic **data** *Proxy a* = *Proxy* definition, the use of a GADT with multiple value constructors helps the type system deduce the types of type-level sets, making our specifications potentially more robust.

```
data F (a :: Fault) where
  MkStarTrackerBatteryError      :: F StarTrackerBatteryError
  MkStarTrackerFixNotFound       :: F StarTrackerFixNotFound
  MkStarTrackerProcessingError :: F StarTrackerProcessingError
```

For convenience, we define a type that makes our signatures slightly shorter:

```
type StarTrackerFaultModel = Set′ [F StarTrackerBatteryError
                                  , F StarTrackerFixNotFound
                                  , F StarTrackerProcessingError
                                  ]
```

We can now express that a transformation is supposed to produce a type-level fault.

```
starTrackerSense′ :: MSF (FaultToleranceT StarTrackerFaultModel m)
                         () [StarInfo]
starTrackerSense′ = starTrackerSense ⋙ arrM
                         (liftUnreliableT (asSet′ [MkStarTrackerBatteryError,
                                                  , MkStarTrackerFixNotFound,
                                                  , MkStarTrackerProcessingError
                                                  ]))
```

We adapt the main control MSF accordingly. Once again, this shows that only the type changes, but not the definition, which is a key strength of this proposal and of MSFs in general.

```
controlSystemStep′  :: Attitude
                    → MSF (FaultToleranceT StarTrackerFaultModel m) () [Action]
controlSystemStep′ initialAttitude = proc (desiredAttitude) → do
  stars         ← starTrackerSense′ ≺ ()
  inertialInfo ← imuSense′          ≺ ()
  attitude      ← feedback initialAttitude (arr (dup ∘ calculateAttitude))
                       ≺ (stars, inertialInfo)
  let actions   = calculateActions desiredAttitude attitude
  returnA ≺ actions
```

9.1 Type-level fault tolerance

In this formalization, recovering from a fault now requires eliminating elements from the fault set. Due to our definition of *Monad* being based on the union of elements, we need to pass the MSFs as arguments to another MSF in the same way we did before. This operation is very similar to the one we defined for value-level sets before except that it uses the API available for type-level sets:

```
starTrackerAvg' :: FaultToleranceT StarTrackerFaultModel m StarInfo
                → FaultToleranceT StarTrackerFaultModel m StarInfo
                → FaultToleranceT StarTrackerFaultModel m StarInfo
                → FaultToleranceT (Set' [ ]) m StarInfo
starTrackerAvg' starInfo1 starInfo2 starInfo3
  = FaultToleranceT $ do
      (f1, d1) ← runFaultToleranceT starInfo1
      (f2, d2) ← runFaultToleranceT starInfo2
      (f3, d3) ← runFaultToleranceT starInfo3
        -- Should be empty
      let fs = (f1 `union` f2 `union` f3) `delete`
                 '[MkStarTrackerBatteryError
                  , MkStarTrackerFixNotFound
                  , MkStarTrackerProcessingError
                  ]
      pure (fs, starTrackerAvg d1 d2 d3)
starTrackerAvg :: StarInfo → StarInfo → StarInfo → StarInfo
starTrackerAvg = ...
```

Note that, in this case, we assume that faults of the same kind but different origins (i.e., from different redundant devices) are represented differently. If this is not the case, then the type should be modified to allow for similar faults from different origins, either with arguments in the type constructor or using a type-level multiset.

The types of MSFs reflect the precise sets of faults that need to be considered. In particular, the compiler would throw an error if we indicate in the type of an MSF that it does not need to consider a fault if an internal MSF does need it. This is a key strength of this proposal: it lets us know, *precisely, what can fail and why*. Together with the system of probabilities introduced before, this allows us to specify not only tolerance ranges of our operations and their probabilities of occurrence, but also the reasons for certain values to be out of their normal ranges and to require specific operations to recover from those failures.

10 Implementation

We have implemented the extension of MSFs in Haskell and in Idris.

Haskell. The implementation in Haskell builds on a larger body of previously existing work, which makes it potentially more useful in practice. Due to the fact that these parameterized *MSF*s subsume the already existing abstraction for MSFs, it makes it possible to define the latter as a special case of the former, making it also possible to test our work with previously existing programs. Furthermore, because Yampa can also be built on top of MSFs, this makes our extension usable for existing work on games and interactive applications.

We have implemented versions of parameterized MSF using both parametric monads (via the `monad-param` library) and effect monads (Orchard & Petricek, 2015) (via the `effect-monad`). The implementation of these functions, as well as any kind of type-dependent programming in Haskell with type-level sets, requires the addition of monoidal laws as type constraints to the types of MSF combinators. In the case of `monad-param`, the type system finds it harder to deduce the types of intermediate monads in do-blocks. The use of additional GHC plugins (Bracker & Nilsson, 2016) facilitates working with these structures.

The features available in Haskell for dependently typed programming are limited and mostly enabled via a series of extensions. Our current definitions make use of *AllowAmbiguousTypes*, *DataKinds*, *FlexibleContexts*, *FlexibleInstances*, *GADTs*, *KindSignatures*, *MultiParamTypeClasses*, *NoMonomorphismRestriction*, *TypeFamilies*, *TypeOperators*, and *RebindableSyntax*.

Perhaps the biggest limitation in practice is the difficulty in Haskell to prove certain trivial laws about sets to fulfill the aforementioned constraints, such as that the empty set is the identity under set union. Effect monads also include the possibility of constraining the monad, and these constraints are included as invariants in bind and, therefore, in every monadic computation that uses bind. Incorporating these constraints makes signatures more complex. We expect some of these limitations to go away as new features of dependently typed programming are introduced in Haskell and they gain more traction among users. These minor nuances do not impact the process of describing reactive systems in practice, and the syntax and support available in Haskell for type-level sets make it relatively convenient to work with this proposal.

The inclusion of type definitions for monoidal constraints on the parameters of effect monads, which we also followed in this paper, makes some constructions harder. In particular, we could have used, as the monoid, functions that alter the sets of faults, with function composition and identity. This would give us a general mechanism to perform fault tolerance, by removing faults without having to encapsulate an *MSF* in another to make it "safe". We have experimented with creating this extension, but the arity of the *Effect* type class and the definition of a proper *Unit* type make this approach unsuccessful. We have found similar problems when working with *constrained monads* as provided by the `supermonads` library (Bracker & Nilsson, 2016).

Idris. We have implemented a basic prototype of MSFs also in Idris, similarly to the one in Haskell. The existence of libraries to work with sets and with probabilities, based on the ones we find in Haskell, make the implementation relatively straightforward.

Idris expects certain functions to be total. Due to our basic type being coinductive and due to the flexibility in the monad, proofs of properties of MSFs and our extensible MSFs would require constraining the monad to be strictly positive. Basic proofs of Arrow laws already exist (Bärenz *et al.*, 2016), and a verified proof of these and other properties remains as future work.

In spite of Idris being a dependently typed language and making it easier to work with values at the type level and to define type functions, similar obstacles were found when trying to define operations that worked on type-level sets of faults. Proofs of some of these laws had to accompany the implementation, and some of the simplest ones (e.g., union with the empty set does not change a set) remain as future work[7]. This is simply a result of our type-level encoding of sets being based on sorted, balanced trees, which makes it harder to write proofs of equality between sets.

11 Related work

In this paper, we have presented a technique to bring fault tolerance information to the types of monadic computations. We have expressed two kinds of fault tolerance information: runtime information and compile-time information. The use of runtime information is more suitable when the information changes over time, or when the type system is not powerful enough to make deductions based on this information. The use of compile-time information is more appropriate for data that do not vary dynamically, and that is simple enough for the compiler to analyze it and manipulate it.

Our work falls in the intersection between fault tolerance, type systems, and functional programming. However, many aspects of fault tolerance are not addressed by this paper. For example, we have not discussed techniques to determine which faults to consider, or how to define and delimit FCRs, or fault detection and masking algorithms, error recovery, or proofs of consensus and agreement. For an in-depth description of the field, see Avižienis (1967, 1976) and Butler (2008).

Functional programming. The use of functional languages in critical systems is not new.

OCaml has been used to implement compilers and tools (Pagano *et al.*, 2009) that comply with strict aviation guidelines (RTCA, 2011).

In the area of Haskell, specifically, we find that the language has been used both for code generation from embedded DSLs and for verification. Copilot (Pike *et al.*, 2010, 2012) is a constrained domain-specific language specify runtime monitors that detect property violations on stream-based C applications. The language is implemented as a Haskell DSL whose execution generates correct C code. This effort is complementary to our proposal, as runtime monitors seek to detect faults during execution (as opposed to specifying them in the types and check them during compilation). In theory, it might be possible to combine both frameworks and provide a variant in which external data streams are tagged with the kinds of unhandled faults that may have affected them.

Huch and Norbisrath introduce an extension to Haskell for distributed programming (Huch & Norbisrath, 2000). This extension provides some elementary facilities to

[7] Idris defines a keyword, *believe_me*, which allows us to defer proving the most basic statements. Care has been taken to minimize the use of this keyword to only laws that we know from other fields to be true. The framework computes the type-level sets of faults correctly, and lack of these proofs does not invalidate the claims of this paper or the usefulness of this work.

tolerate specific kinds of faults in message passing applied to distributed networks. The definition of fault tolerance that Huch and Norbirath use is, however, restricted to software implementations, and much narrower than the one presented in Avižienis (1967). In particular, it does not consider general fault detection and masking, the definition of FCRs, the specification of clear fault models, or the use of type systems to guarantee compliance with a fault model. Cloud Haskell (Epstein *et al.*, 2011) includes primitives to handle fail-stop faults in software processes, similar to the *supervisor* mechanism provided by Erlang (Vinoski, 2007), but does not use type-level programming to capture properties of the type model and guarantee that all the faults in the fault model are handled properly. Glasgow distributed Haskell (GdH) is a Haskell extension for distributed systems (Trinder *et al.*, 2000). The authors do identify the same key aspects of fault tolerance that we work with, but they only deal with runtime aspects and do not use the type system to detect potential faults. Furthermore, the way that some faults are addressed (re-execution) does not guarantee that the system tolerates that class of faults, and at best transforms fail-stops into delays. Depending on the timing requirements of the system built on top of GdH, long delays might still constitute serious faults. This makes the proposed approach limited in its fault tolerance, and further analysis and a clear fault model are needed. Haskell distributed parallel Haskell (HdpH) (Stewart, 2013; Stewart *et al.*, 2013; Maier *et al.*, 2014; Stewart *et al.*, 2016) is a variant of distributed parallel Haskell for reliable computation. HdpH does provides monitoring and recovering capabilities but, like other proposals for distributed Haskell, its fault tolerance mechanisms are applied during runtime, not during compile time, and it does not provide a mechanism to verify the correct handling of fault classes.

Stream programming in critical systems. Synchronous dataflow programming languages have been used to specify well-formed stream-based mission critical systems. Esterel (Boussinot & De Simone, 1991) is a synchronous language with formally verified semantics that has been used in avionic software development (Berry *et al.*, 2000). Scade (Dormoy, 2008) is a language and set of tools based on Lustre (Halbwachs *et al.*, 1991) that provides static analysis and verification. These languages are based on an underlying notion of stream programming, which is complementary to our proposal. Just like with Copilot, these synchronous languages could be extended with a notion of explicit type-level fault tolerance.

Monads and effects. Types like *Maybe*, *Either* and list have been used often to capture failures or exceptions in lazy functional languages (Spivey, 1990; Wadler, 1985). The difference between these types and our proposal is that these types assume that failures can be detected, whereas we do not work under that assumption. Nevertheless, there is a class of failures, termed *omissive* or *benign*, defined as those that can be detected as failures, and key in the design of fault-tolerant systems is to add redundancy to make some undetectable or *transmissive* faults detectable or *omissive*. The combination of an exception monad with a fault tolerance monad could provide a very explicit description of all the faults that are not handled, and, of those that are handled, a partitioning between transmissive and omissive failures.

Mechanisms like *extensible effects* (Kiselyov *et al.*, 2013) have been proposed as a composable alternative to monad transformers (Liang *et al.*, 1995). Their use and, especially, the possibility of combining different kinds of effects irrespective of their stacking order might ease the specification of mechanisms for fault tolerance at different architectural levels. We have explored the use of effect monads or graded monads (Orchard *et al.*, 2014), polymonads (Bracker & Nilsson, 2015), and supermonads (Bracker & Nilsson, 2016). We have not explored the use of unconstrained indexed monads (Katsumata, 2014; Orchard & Petricek, 2014). The use of some of these constructs could also help represent MSFs that account for the duration of processing a value, or that contain type-level information on how much history of a stream must be kept in memory. Alternatives can also be found in dependently typed languages like Idris, with different proposals to compose effects (Brady, 2013).

Formal methods in critical systems. The Prototype Verification System (PVS) is a dependently typed specification language and an associated theorem prover (Owre *et al.*, 1992, 1996, 1999). PVS has been used in multiple critical systems and fault tolerance (Rushby, 2006). To name a few areas of application, PVS has been used to verify interactive consistency algorithms that handle Byzantine faults (Lincoln & Rushby, 1993, 1994) and clock synchronization algorithms (Pfeifer *et al.*, 1999). In aerospace, an area that is heavily regulated, PVS has been used for fault tolerance in aircraft control systems (RW, 1996; Dutertre & Stavridou, 1997; Di Vito, 1999), and manned and interplanetary spacecrafts (Di Vito, 1996; Di Vito & Roberts, 1996; Crow & Di Vito, 1998). To the best of our knowledge, no work attempts to use PVS's type system to capture unhandled faults or provide a partitioning of the fault space based on the fault tolerance mechanisms of the subsystems used.

Formal verification has also been used to prove properties of seL4 (Klein *et al.*, 2009, 2014), an L4 microkernel implementation that has been verified in Isabelle/HOL. While the goal of seL4 is to provide a kernel that is proven free from programming errors, it has been extended with a fault-tolerant real-time scheduler (Xu *et al.*, 2016) that handles some timing faults caused by hardware or software failures.

Our work presents methods to compute reliability factors analytically, and also test and verify systems as a black box, with a focus on producing systems that are proven correct by design. The calculation of actual reliability factors based on the errors produced by a system during testing has been deemed misleading (Butler & Finelli, 1993), due to inaccuracies in the results and the large number of tests needed to verify ultra-reliability. Earlier examples in which reliability factors are added as part of a fault tolerance monad currently work only for systems in different FCRs or without common cause faults. The approach to testing presented in Section 7 can be adapted to introduce common cause faults, but it does not otherwise reduce the large number of tests needed to explore to verify ultra-reliable systems.

12 Future work

This work has presented a simple approach at capturing aspects of fault tolerance in reactive and FRP Haskell. We have shown how to tag existing reactive transformations with reliability factors, with probabilities and with a sense of the possible unhandled faults that may have affected a result. We have demonstrated this approach by averaging the results from multiple sensors, thus increasing the reliability of the results. We have indicated how these fault tolerance mechanisms can be evaluated in a systematic manner, by capturing system properties using temporal logic and by injecting different faults at different points in the network. We have also shown that, with dependently typed programming, we can move some of this information to the type level and carry out compile-type analysis of the faults that may affect a reactive system. We have given a schematic definition of an extended reactive framework to handle parameterized monads, which we have implemented also using different monadic extensions in Haskell and in Idris. This extension does not require adaptations to handle fault tolerance, and all the proposals explored in this paper are defined orthogonally in separate monads, which is a key strength of our proposal.

We are currently investigating if it would be possible to use this proposal to describe some of the control systems in small, unmanned aircrafts. While the use of Haskell itself during runtime may not be tolerable due to the lack of real-time guarantees imposed by the garbage collector, compiling a reactive network to C or another language may produce better results. The commonalities between FRP frameworks, MSFs, and stream-based systems like Copilot (Pike *et al.*, 2010) suggest that there may be an underlying abstraction that could be used to represent time-varying systems more generally and later verify, test, simulate or compile as desired. Also, using a stream framework in which data processors can be replaced during execution would facilitate using QuickCheck to study the behavior of the system when internal components malfunction.

Our use of reliability factors was done at execution level, but we have recently successfully used a similar approach to add rational numbers at the type level and obtain static information about reliability, and a compile-time understanding of the probabilities of different kinds of failures. The use of a dependently typed language, like Idris, would facilitate operating on rational numbers at the type level, and is obtaining compile-time reliability factors.

The fault tolerance mechanisms proposed all work similarly: faults are reversed (i.e., reliability factors are inverted, faults removed from sets, etc.). All the proposals included in this paper can be expressed more generally in terms of noncommutative rings. We expect this to let us generalize the recovery and fault tolerance mechanisms. A common mechanism in fault tolerance is the re-execution of transactions when failures are detected, something that monads like *STM* already provide. In terms of fault tolerance, we have left aside the study of faults that are related to other faults, as well as timing faults. Dependent probabilities can already be expressed in our proposal. Nevertheless, we expect to explore this further in the future, in combination with ongoing work on MSFs with type-level clocks (Bärenz & Perez, 2018) and integration with QuickCheck.

Acknowledgments

The authors thank Paul Miner, Wilfredo Torres, Kerianne Hobbs, Frank Dedden, and William Edmonson for multiple discussions on fault tolerance. The authors also thank Swee Balachandran for suggestions and comments on earlier versions of this paper, anonymous reviewers for their detailed feedback during numerous reviews, and the journal and special issue editors. This material is based upon work funded under NASA Cooperative Agreement 80LARC17C0004.

Conflict of interest

There is no conflict of interest to declare.

References

Avižienis, A. (1967) Design of fault-tolerant computers. In Proceedings of the November 14-16, 1967, Fall Joint Computer Conference, AFIPS ’67 (Fall). New York, NY, USA: ACM, pp. 733–743.

Avižienis, A. (1976) Fault-tolerant systems. *IEEE Trans. Computers* **25**(12), 1304–1312.

Bärenz, M. & Perez, I. (2018) Rhine: FRP with type-level clocks. In Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell, Haskell 2018. New York, NY, USA: ACM, pp. 145–157.

Bärenz, M., Perez, I. & Nilsson, H. (2016) *Mathematical properties of monadic stream functions*. Available at: http://cs.nott.ac.uk/∼ixp/papers/msfmathprops.pdf

Bedingfield, K., Leach, R. D. & Alexander, M. B. (1996, August) *Spacecraft System Failures and Anomalies Attributed to the Natural Space Environment*. Technical report, National Aeronautics and Space Administration. NASA Reference Publication 1390.

Berry, G., Bouali, A., Fornari, X., Ledinot, E., Nassor, E. & De Simone, R. (2000) Esterel: A formal method applied to avionic software development. *Sci. Comput. Program.* **36**(1), 5–25.

Boussinot, F. & De Simone, R. (1991) The ESTEREL language. *Proc. IEEE* **79**(9), 1293–1304.

Bracker, J. & Nilsson, H. (2015) Polymonad programming in Haskell. Proceedings of the 27th Symposium on the Implementation and Application of Functional Programming Languages, IFL ’15. New York, NY, USA: ACM, pp. 3:1–3:12.

Bracker, J. & Nilsson, H. (2016) Supermonads: One notion to bind them all. Proceedings of the 9th International Symposium on Haskell, Haskell 2016. New York, NY, USA: ACM, pp. 158–169.

Brady, E. (2013) Programming and reasoning with algebraic effects and dependent types. *ACM SIGPLAN Not.* **48**, 133–144.

Butler, R. W. (2008, February). *A Primer on Architectural Level Fault Tolerance*. Technical report. NASA/TM-2008-215108, L-19403. NASA Langley Research Center.

Butler, R. W. & Finelli, G. B. (1993) The infeasibility of quantifying the reliability of life-critical real-time software. *IEEE Trans. Softw. Eng.* **19**(1), 3–12.

Butler, R. W. (1996) *An Introduction to Requirements Capture Using PVS: Specification of a Simple Autopilot*. Technical report.

Courtney, A., Nilsson, H. & Peterson, J. (2003) The Yampa arcade. Haskell Workshop, pp. 7–18.

Crow, J. & Di Vito, B. (1998) Formalizing Space Shuttle software requirements: Four case studies. *ACM Trans. Softw. Eng. Methodol. (TOSEM)* **7**(3), 296–332.

Di Vito, B. L. (1996) Formalizing new navigation requirements for NASA’s space shuttle. In International Symposium of Formal Methods Europe. Springer, pp. 160–178.

Di Vito, B. L. (1999) A model of cooperative noninterference for integrated modular avionics. *Depend. Comput. Crit. Appl.* **7**, 269–286.

Di Vito, B. L. & Roberts, L. W. (1996) *Using Formal Methods to Assist in the Requirements Analysis of the Space Shuttle GPS Change Request*. Technical report.

Dormoy, F.-X. (2008) Scade 6: A model based solution for safety critical software development. Proceedings of the 4th European Congress on Embedded Real Time Software (ERTS 2008), pp. 1–9.

Driscoll, K. (2008) *Real system failures*. https://c3.nasa.gov/dashlink/static/media/other/ObservedFailures4.html

Dutertre, B. & Stavridou, V. (1997) Formal requirements analysis of an avionics control system. *IEEE Trans. Softw. Eng.* **23**(5), 267–278.

Elliott, C. & Hudak, P. (1997) Functional reactive animation. *ACM SIGPLAN Not.* **32**(8), pp. 263–273.

Epstein, J., Black, A. P. & Peyton-Jones, S. (2011) Towards Haskell in the cloud. *ACM SIGPLAN Not.* **46**, pp. 118–129. ACM.

Erwig, M. & Kollmansberger, S. (2006) Functional pearls: Probabilistic functional programming in Haskell. *J. Func. Program.* **16**(1), 21–34.

Halbwachs, N., Caspi, P., Raymond, P. & Pilaud, D. (1991) The synchronous data flow programming language LUSTRE. *Proc. IEEE* **79**(9), 1305–1320.

Huch, F. & Norbisrath, U. (2000). Distributed programming in Haskell with ports. In Symposium on Implementation and Application of Functional Languages. Springer, pp. 107–121.

Hughes, J. (2000) Generalising monads to arrows. *Sci. Comput. Program.* **37**(1), 67–111.

Katsumata, S.-y. (2014) Parametric effect monads and semantics of effect systems. *ACM SIGPLAN Not.* **49**(1), 633–645.

Kiselyov, O., Sabry, A. & Swords, C. (2013) Extensible effects: An alternative to monad transformers. *ACM SIGPLAN Not.* **48**, 59–70.

Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., *et al.* (2009) seL4: Formal verification of an OS kernel. In Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles. ACM, pp. 207–220.

Klein, G., Andronick, J., Elphinstone, K., Murray, T., Sewell, T., Kolanski, R. & Heiser, G. (2014) Comprehensive formal verification of an OS microkernel. *ACM Trans. Comput. Syst. (TOCS)*, **32**(1), 2.

Kshemkalyani, A. D. & Singhal, M. (2008) *Distributed Computing*. Cambridge University.

Lamport, L., Shostak, R. & Pease, M. (1982) The Byzantine generals problem. *ACM Trans. Program. Lang. Syst. (TOPLAS)* **4**(3), 382–401.

Liang, S., Hudak, P. & Jones, M. (1995) Monad transformers and modular interpreters. In Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. ACM, pp. 333–343.

Lincoln, P. & Rushby, J. (1993) A formally verified algorithm for interactive consistency under a hybrid fault model. *The Twenty-Third International Symposium on Fault-Tolerant Computing, 1993. FTCS-23. Digest of Papers.* IEEE, pp. 402–411.

Lincoln, P. & Rushby, J. (1994) Formal verification of an interactive consistency algorithm for the Draper FTP architecture under a hybrid fault model. In In Proceedings of the Ninth Annual Conference on Computer Assurance, 1994. COMPASS 1994 Safety, Reliability, Fault Tolerance, Concurrency and Real Time, Security. IEEE, pp. 107–120.

Maier, P., Stewart, R. & Trinder, P. (2014) The HdpH DSLs for scalable reliable computation. In Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell, Haskell 2014. New York, NY, USA: ACM, pp. 65–76.

Nilsson, H., Courtney, A. & Peterson, J. (2002) Functional reactive programming, continued. In Haskell Workshop, pp. 51–64.

Orchard, D. & Petricek, T. (2014) Embedding effect systems in Haskell. In Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell, Haskell 2014. New York, NY, USA: ACM, pp. 13–24.

Orchard, D. & Petricek, T. (2015) Embedding effect systems in Haskell. *ACM SIGPLAN Not.* **49**(12), 13–24.

Orchard, D. A., Petricek, T. & Mycroft, A. (2014) The semantic marriage of monads and effects. *CoRR*, abs/1401.5391.

Owre, S., Rushby, J. M. & Shankar, N. (1992) PVS: A prototype verification system. In *11th International Conference on Automated Deduction (CADE)*, Kapur, D. (ed.) Lecture Notes in Artificial Intelligence, vol. 607. Saratoga, NY: Springer-Verlag, pp. 748–752.

Owre, S., Rajan, S., Rushby, J. M., Shankar, N. and Srivas, M. (1996) PVS: Combining specification, proof checking, and model checking. In International Conference on Computer Aided Verification. Springer, pp. 411–414.

Owre, S., Shankar, N., Rushby, J. M. & Stringer-Calvert, D. W. J. (1999) *PVS Language Reference*. Technical report.

Pagano, B., Andrieu, O., Moniot, T., Canou, B., Chailloux, E., Wang, P., Manoury, P. & Colaço, J.-L. (2009) Experience report: Using objective caml to develop safety-critical embedded tools in a certification framework. In Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming, ICFP 2009. New York, NY, USA: ACM, pp. 215–220.

Perez, I. (2017) Back to the future: Time travel in FRP. Proceedings of the 10th ACM SIGPLAN International Haskell Symposium, Haskell 2017. New York, NY, USA: ACM.

Perez, I. (2018) Fault-tolerant functional reactive programming (functional pearl). *Proc. ACM Program. Lang.*, **2**(ICFP), 96:1–96:30.

Perez, I. & Nilsson, H. (2017) Testing and debugging functional reactive programming. *Proc. ACM Program. Lang.* **1**(ICFP), 2:1–2:27.

Perez, I. & Nilsson, H. Runtime verification and validation of functional reactive systems. *J. Func. Program*. Submitted (Under evaluation).

Perez, I., Bärenz, M. & Nilsson, H. (2016) Functional reactive programming, refactored. In Proceedings of the 9th International Symposium on Haskell, Haskell 2016. New York, NY, USA: ACM, pp. 33–44.

Perez, I., Goodloe, A. & Edmonson, W. (2019) Fault-tolerant swarms. In 7th IEEE International Conference on Space Mission Challenges for Information Technology (SMC-IT 2019). IEEE.

Pfeifer, H., Schwier, D. & Von Henke, F. W. (1999) Formal verification for time-triggered clock synchronization. *Depend. Comput. Crit. Appl.* **7**, 207–226.

Pike, L., Goodloe, A., Morisset, R. & Niller, S. (2010). Copilot: A hard real-time runtime monitor. In International Conference on Runtime Verification. Springer, pp. 345–359.

Pike, L., Niller, S. & Wegmann, N. (2012) Runtime verification for ultra-critical systems. In *Runtime Verification*, Khurshid, S. & Sen, K. (eds), Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 310–324.

RTCA. (2011) *Software Considerations in Airborne Systems and Equipment Certification (178C)*. Technical report.

Rushby, J. (2006) *PVS Bibliography*. http://pvs.csl.sri.com/documentation.shtml

Spivey, M. (1990) A functional theory of exceptions. *Sci. Comput. Program.* **14**(1), 25–42.

Stewart, R. (2013) *Reliable Massively Parallel Symbolic Computing: Fault Tolerance for a Distributed Haskell*. Ph.D. thesis, Heriot-Watt University.

Stewart, R., Trinder, P. & Maier, P. (2013) Supervised workpools for reliable massively parallel computing. In *Trends in Functional Programming*, Loidl, H.-W. & Peña, R. (eds), Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 247–262.

Stewart, R., Maier, P. & Trinder, P. (2016) Transparent fault tolerance for scalable functional computation. *J. Func. Program.* **26**.

Trinder, P., Pointon, R. & Loidl, H.-W. (2000) Towards runtime system level fault tolerance for a distributed functional language. *Trends Func. Program.* **2**, 103.

Vinoski, S. (2007) Reliability with Erlang. *IEEE Internet Comput* **11**(6).

Wadler, P. (1985) How to replace failure by a list of successes. In Conference on Functional Programming Languages and Computer Architecture. Springer, pp. 113–128.

Xu, L., Bai, Y., Cheng, K., Ge, L., Nie, D., Zhang, L. & Liu, W. (2016) Towards fault-tolerant real-time scheduling in the seL4 microkernel. In 2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS). IEEE, pp. 711–718.