

Modular Architectural Representation and Analysis of Fault Propagation and Transformation

Malcolm Wallace

*Dept of Computer Science,
University of York, UK*
Malcolm.Wallace@cs.york.ac.uk

Abstract

This paper describes a modular representation and compositional analysis of a system's hardware and software components, called Fault Propagation and Transformation Calculus (FPTC). We show, given an architectural description of how components are combined into a whole system, together with an FPTC expression of each component's failure behaviour, how the failure properties of the whole system can be computed automatically from the individual FPTC expressions.

From a safety point of view, this provides some idea of robustness: the system's capability to withstand certain types of failures in individual components. It also provides a way to understand how and where to develop fault accommodation within an architecture.

Keywords: components, architecture, safety-critical, validation

1 Introduction and Motivation

High integrity real-time software systems (HIRTS), as the name implies, have substantial requirements for reliability and safety. Traditional software analysis techniques tend towards determining a product's correctness, whether through proof, model-checking, abstract interpretation, or some other formal method. However, in the safety domain, it is of equal interest to know how a system behaves in the presence of failure, regardless of whether that failure is in the external environment, or caused by an internal software error. A demonstrable ability to continue to function in the presence of failures is good, and methods to mitigate potential hazards are often mandated by certification

authorities.

But, in the main, analysis of the safety properties of software is manual, and therefore expensive. What is more, the analysis is not modular or compositional. This means that any change to even the smallest component may force a complete re-analysis of the whole system – there is currently no way to determine the locus of impact, in safety terms, of the change. This has huge cost implications for development in areas like incremental certification, product line families, and so on.

In this paper we present a modular representation of software architecture for HIRTS that focusses on component failure properties, (rather than the expected, correct behaviour). It is much easier for an engineer to analyse an individual component manually for its local failure behaviour, than to attempt to analyse an entire system. The failure behaviour of components is then recorded in a fully compositional manner, allowing the automatic determination of the failure behaviour of the whole system. To this end, a semantics of failure propagation and transformation is outlined, which we name FPTC (Failure Propagation and Transformation Calculus). Thus, the potential of automatic analysis to reduce the cost of change is enormous: if the fault behaviour of the system as a whole is calculated purely from the composition of its parts, then the impact of a component change or architectural change can be predicted very cheaply.

Furthermore, if unexpected failure behaviour emerges at the system level of the model, we claim it will be possible to quickly trace where its causes lie. Thus, one can determine more readily where fault accommodation patterns must be added to the architecture to mitigate or correct potential failures.

In this paper, we restrict our attention mainly to software. However we hope that the techniques presented might be capable of adaptation to complete engineering systems, including the physical hardware, electrical and hydraulic plant, and so on. Also, whilst our focus on failure properties does not address the full range of safety concerns, it does however provide one important piece of evidence towards the construction of a safety argument.

2 Architectural Descriptions of Software

In order to model the failure behaviour of a system in a compositional manner, what kind of architectural model do we need to build first? What descriptions of components, and the connections between them, are of best utility? Indeed, taking a step further back what entities comprise a component or a connection?

In the hard real-time software domain, we believe that the minimum useful unit of architecture – the building-block – is the statically schedulable code

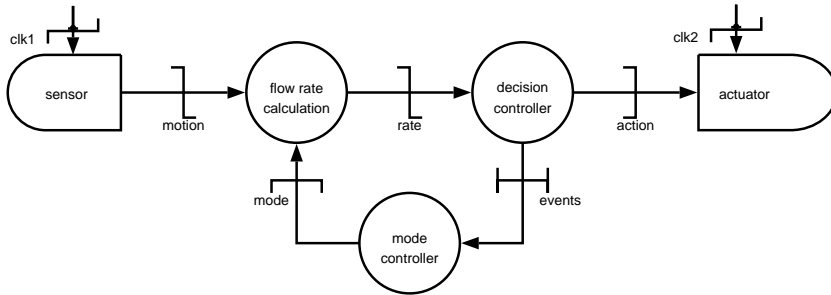


Figure 1. A simple example of a Real-Time Networks architecture.





	destructive read (blocking)	non-destructive read (non-blocking)
non-destructive write (blocking)	 channel	 constant
destructive write (non-blocking)	 signal	 pool

Figure 2. The basic RTN communications protocols.

unit, whether periodic or event-driven. This unit represents a single sequential thread of control. It may be executed many times in response to the stimulus of either a clock or an external event. The basic connections between units are therefore the communications protocols between these threads. Communication may take the form of handshakes, buffers, prods, and so on, each of which captures a slightly different data and control flow behaviour. Each protocol will be modelled separately, since each may have different failure properties.

We assume that in a real-time system, the software architecture is statically determined; that is, all threads and communications channels are known to the designers before initialisation, not created and destroyed dynamically through the lifetime of the computation. We also assume that a rate monotonic (or other) static analysis has been performed to determine whether the collection of threads is schedulable on given processor hardware.

However, even a static schedule can allow certain timing fluctuations (jitter, etc.) to occur, and thus we can choose to model architectural timing dependencies as well.

The final element of the model, should the system be spread over several physical processing elements, is the allocation of threads to processors and of comms to network elements. The corruption of one physical element can propagate a failure to all the software running on it (or in the case of a network or bus, the communications running through it).

These requirements for an architectural description lead to a design like the

Real-Time Networks (RTN) formalism [8], based on the MASCOT notation [9]. Essentially, RTN describes a graph of nodes and arcs: nodes represent hardware (sensors, actuators) or schedulable code units, and arcs represent communications protocols, with each arc being annotated with the protocol type. Figure 1 illustrates a simple RTN graph, with a key to the protocols in Figure 2. RTN has tool support to refine these designs hierarchically, to formally define the code units as state machines, and to auto-generate Ada code from the design.

3 Basic Failure Modelling

3.1 Notation

Each component of the system can be analysed in isolation from the rest of the system (by experts), for its failure behaviour in response to potential failure stimuli. All possible failures on input must be considered.

We follow here the conventional HAZOP/SHARD [6] identification of *types* of failures through a set of *guidewords*, such as: value failures (detectably wrong, undetectably wrong, stale); timing failures (early, late); and sequence failures (omission, commission). This set of guidewords is not comprehensive, and can be tailored to the failures of interest in a particular setting. The failure types are expressed for simplicity as a simple enumerated type – a more structured sum-of-products type could be useful in a more nuanced setting with multiple levels of detail about failure types, such as modelling the severity of a failure. Crucially, “normal” behaviour is a natural member of the enumerated type, representing the lack of a failure. (Intuitively this, like 0 in the natural numbers, is the ‘zero’ in the algebra of failures.)

The behaviour of a component may be propagational (passing on a failure from inputs to outputs), or transformational (changing the nature of the failure from one type to another). Of course, a component may also be a source or sink of failures – a source, if the component itself may introduce a failure without external stimulus, and a sink, if the component is capable of detecting and correcting a failure propagated from elsewhere in the system.

These different responses to input can be captured in a simple first-order functional pattern-based notation. Using $*$ to indicate normal (no failure) behaviour, the following four expressions denote some example source, sink, propagation, and transformation responses, respectively, for a simple 1-input 1-output component:

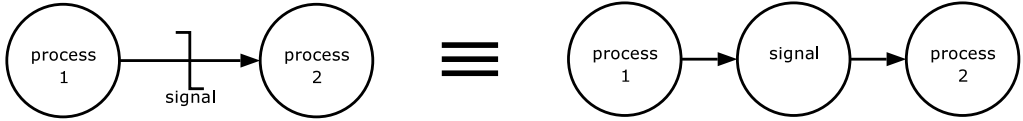


Figure 3. Re-modelling a comms protocol as a component. Because comms protocols embody both control flow and data flow, they can introduce or transform failure types, and so must be assigned FPTC expressions just like components.




		
signal	channel	pool
early -> * omission -> late commission -> value	early -> * omission -> late commission -> late * -> late	early -> * omission -> stale value commission -> * late -> stale value

Figure 4. FPTC expressions for the common RTN comms protocols. It may initially be surprising that a channel transforms a commission fault to a late timing fault, but this is due to the writer blocking until the extra data value has been processed by the reader, which may then lead to a delay in processing subsequent messages.

*	→	late	(source)
early	→	*	(sink)
omission	→	omission	(propagate)
late	→	value	(transform)

Typically, a component is actually modelled by a number of pattern/result expressions (*transforms*, or *clauses*) of this form, and the cumulative effect is its overall *behaviour*. A more detailed exposition of the complete expression language is given in Section 4.

It is important to understand how failure transformation, as in the last clause of the example, can arise. Take for instance a pool-like comms protocol: the most recently written value is retained in a buffer which can be non-destructively read by the receiver, but destructively overwritten by the supplier. Thus, if the supplier is late writing to the buffer, the reader is not delayed, but may instead proceed with a stale value. As another example, consider a dual-lane voting system with identical lanes. If a random hardware error causes one lane to receive an undetectably wrong value, the voter component can convert it to a detectable value failure, through comparison with the other lane. With three or more lanes, the voter can instead act as a sink, not only detecting the fault but also correcting (or compensating) for it.

To model the system as a whole, every node of the architecture graph is assigned an FPTC behaviour. Now, because each communications protocol has its own potential failure propagation behaviours, a comms protocol must be treated identically in this framework to the computational components

$$\text{stabilise}(\text{graph}) = \text{fix}(\lambda g \rightarrow \forall(c :: \text{Component}) \in g \bullet \text{out}(c)) \text{ graph}$$

$$\text{fix}(f) x = \begin{cases} x, & \text{if } x = f(x) \\ \text{fix}(f) f(x), & \text{otherwise} \end{cases}$$

$$\text{in}(c) = \bigcup_{p \in \text{pred}(c)} \text{out}(p)$$

$$\text{out}(c) = \bigcup_{t \in \text{transforms}(c), f \in \text{in}(c)} \mathcal{A}[[t]] f$$

$$\mathcal{A}[[* \rightarrow f]]_- = \{f\}$$

$$\mathcal{A}[[f \rightarrow *]]_- = \emptyset$$

$$\mathcal{A}[[f \rightarrow g]] f' = \begin{cases} \{g\}, & \text{if } f = f' \\ \emptyset, & \text{otherwise} \end{cases}$$

Figure 5. The propagation and transformation algorithm. It stabilises on the maximal tokenset of each connection in the graph of components, by calculating the fixpoint of the dataflow equations for *in* sets and *out* sets. At initialisation, all sets contain only the “normal behaviour” token. The *out* set of each component is calculated through an evaluation function \mathcal{A} , which applies the pattern-matching transformation clauses to the *in* set, thus augmenting the output tokenset with the results. The output tokenset is plumbed by the graph connectivity to the inputs of other components. The rules for *in* and *out* sets are ordinary dataflow equations.

of the system, that is, it too must be given an FPTC behaviour. Figure 3 illustrates rather trivially how a comms connection between two components can be re-considered as a system of three components, and Figure 4 gives the failure behaviours of the main RTN protocols. Hence, the arrows connecting the final components together are now free of any failure semantics at all – a useful property.

We also note in passing that there are other types of failure dependency between components, such as the allocation of schedulable software units to CPUs. This kind of dependency can and should be added into the graph as well. For instance, each CPU can be modelled as a component with an FPTC expression describing its failure modes, and connected by error-flow arcs directly to the software processes running on it.

3.2 Semantics

So, how are failure transformation expressions used compositionally to determine the failure properties of the system as a whole? First, we observe that the architectural graph, as modified to ensure that arrows are failure-free, can be

regarded as a token-passing network. Tokens corresponding to failure modes are introduced by sources, transformed and propagated by the network nodes, and removed by sinks. We are interested in the *flows* of failures, and thus can annotate the arrows of the diagram with *tokensets*, that is, the set of all possible failures that can be propagated along a dependency path.

Ultimately, we want to know the maximal tokenset on any flow in the network, and this can be calculated by a straightforward fixpoint algorithm (sketched formally in Figure 5). Starting with the singleton set containing the $*$ token ‘no failure’ as a label on every connection arrow, the expression at every component node is ‘run’, using the tokensets on input arrows as the inputs to the FPTC expressions. The output failure tokens of each component are accumulated on the outgoing arrows, and the system continues to run until a fixed point is reached, that is, no further change is observed in the gathered tokensets. The algorithm is very similar to well-known compiler optimisation techniques for liveness analysis of temporaries during register allocation [1].

This algorithm is guaranteed to terminate, even in the presence of cycles in the connection graph, because there is a finite domain of failure types (tokens), and new tokens can be added to tokensets, but none is ever removed. The algorithm is also convergent, meaning that it does not matter in what order the token transformation expressions are evaluated, the result will always be the same. This is because the expressions being evaluated at each node are purely functional – there is no non-determinism. Finally, the algorithm is stable – it does not flip between states, failing to reach a single fixpoint. This is again due to accumulating tokens on the arrows: transformation expressions are run on all available input failure types – there is no selection or decision between them – and hence the output tokenset can only grow, never diminish. In short, the system is monotonic and decidable.

4 Definition of the Transformation Language

We now turn our attention to the detailed design and specification of the fault propagation and transformation language. Our goals are for the language to be:

- expressive of the problem domain;
- concise, to enable the safety analyst to concentrate on the system at hand, rather than the notation;
- easy to read and write;
- hard to mis-understand, unambiguous.

The last point is especially important in a safety-related analysis, since ambiguity might mislead the system’s designer or certification authority to wrongly accept a hazardous design.

4.1 Default Clauses

A failure transformation model of a component is a series of clauses in the language already sketched above (Section 3.1). Consider the following example, with three clauses describing a single component:

$$\begin{array}{lll} * & \rightarrow & \text{late} \\ \text{omission} & \rightarrow & \text{late} \\ \text{value} & \rightarrow & \text{value} \end{array}$$

What can we say about this component’s behaviour in the presence of, say, a late timing failure on its input? The patterns enumerated on the left of the expressions are not comprehensive. Now, whilst the safety analyst must surely consider every possible input failure condition, it is not necessary for all of these to be notated if we allow some default interpretation of a ‘missing’ clause. After all, indicating the default cases explicitly all the time is (a) tedious and error-prone for the writer, and (b) tends to obscure the non-default cases from the human reader. The choice of default interpretation is either to assume the best, or assume the worst. That is, an omitted pattern for failure type f could either mean that f is always detected and corrected, or it could mean that f is always propagated.

We choose the latter interpretation (assume the worst). Failure propagation is likely to be the common behaviour of any software component in the absence of further design work. But if a component can detect and correct a failure, that is unusual, and should be noted explicitly.

$$\text{transforms}(c) = \text{explicitClauses}(c) \cup \{\forall f \notin \text{patterns}(c) \bullet \llbracket f \rightarrow f \rrbracket\}$$

4.2 Multiple Connections

Our notation so far only appears to deal with components with a single input and single output. This is very simple and clear, but is it adequate for realistic systems, where components almost certainly have multiple input and output connections? Here we explore some of the design choices for expanding the notation to multiple inputs and outputs.

4.2.1 Implicit Multiplexing.

The simplest approach is to implicitly combine multiple connections into the single available position in the expression. That is, on the LHS, a pattern

matches a failure on *any* of the multiple inputs, and on the RHS, a failure is propagated to *all* of the multiple output connections. In some simple systems this will be adequate, and it certainly keeps the notation concise, but in general it will be too pessimistic, propagating too many faults along too many paths.

4.2.2 Multiple Outputs.

Multiple output connections can be handled very straightforwardly by the use of *tuples* in the transformation expression. Each item in the tuple corresponds to a separate output channel, defined positionally. For instance, this expression:

$$\text{late} \rightarrow (\text{value}, *, \text{late})$$

belongs to a component with a single input and three outputs. A timing failure (*late*) on input leads to a *value* failure on the first output, together with a propagation of the *late* timing failure on the third output, but no detectable error on the middle output.

Default clauses also remain simple here, with any input failure f not matched on the left being propagated to *all* output connections on the right.

$$\begin{aligned} \text{transforms}(c) = & \text{explicitClauses}(c) \cup \\ & \{\forall f \notin \text{patterns}(c) \bullet \llbracket f \rightarrow (f, f, \dots f)^n \rrbracket, n = \text{arity}(c)\} \end{aligned}$$

4.2.3 Multiple Inputs.

On the face of it, multiple input connections look as if they can be handled in an identical manner to multiple outputs by tupling the left-hand pattern.

A LHS pattern containing multiple failure tokens implies a manual analysis of the case where multiple errors happen simultaneously on different inputs to a component. Of course, this is entirely reasonable as a requirement, although probably more difficult to determine in practice.

However, there are numerous syntactic subtleties to appreciate here.

Exponential growth in the number of possible clauses.

For example, a component with 5 inputs in a setting where 6 failure modes (+1 non-failure) are being examined, has $7^5 = 16807$ possible LHS patterns. The vast majority of these will have RHS that are either identical or fall into a small number of groups, so we need some way to aggregate similar patterns into a single clause.

Keeping in mind the goals of an easy-to-read yet hard-to-misunderstand notation, we introduce two independent new kinds of pattern: wildcards and variables. A wildcard pattern means we don't care what type of failure (or

non-failure) token occurs in a certain position, and a variable means something similar, except that its value is bound for use on the RHS of the expression. For instance:

$$\begin{aligned}(\text{late}, _) &\rightarrow (\text{value}, \text{late}) \\ (\text{late}, f) &\rightarrow (f, \text{late})\end{aligned}$$

These two expressions mean, respectively, that when a late failure occurs on the first input: (1) it does not matter whether there is a failure or not on the second input; or, (2) whatever the simultaneous failure on the second input, it is propagated to the first output.

Default clauses.

If no clause matches, is it still correct to assume that all input failure modes not explicitly mentioned cascade outwards to all output arrows? Perhaps it is better to give an explicit default, as a ‘catch-all’ clause, e.g.

$$(f, g) \rightarrow (f, \{f, g\})$$

We see immediately that another extension to the notation is necessary, namely the introduction of *sets* of output failures, where previously only single failures were noted. Alternatively, the same effect can be achieved by permitting several default clauses with identical LHS but differing RHS, such as

$$\begin{aligned}(f, g) &\rightarrow (f, f) \\ (f, g) &\rightarrow (f, g)\end{aligned}$$

where the interpretation is that both clauses match in the default case, and so both RHS are generated.

Overlapping patterns.

Any use of a wildcard or variable pattern has the possibility of overlapping with a more specific pattern, e.g.

$$\begin{aligned}(*, \text{late}) &\rightarrow (*, *) \\ (*, f) &\rightarrow (*, f)\end{aligned}$$

Taking these clauses together, the reader might be unsure whether a late failure on the second input is propagated or not – the first clause says no, but the latter says yes. More extremely, an explicit default clause of the form suggested above has a pattern that overlaps with *all* other clauses that might be written! Thus, we need to impose a standard interpretation rule, such as the order in which clauses are written, or a partial ordering from more-specific

to more-general. We prefer the latter rule based on generality/specificity, since the former rule would allow the semantics of the specification to change depending on the textual ordering of clauses, which is somewhat fragile and open to cut-and-paste mistakes. It is of course useful for the analysis toolset to report (optionally) on which clauses are overlapping, to avoid unintentional slips; likewise if any potential patterns are missing, such as when an explicit default is omitted.

Nevertheless, despite the extra semantic complexity involved, it has been demonstrated in practice by well-established programming languages (e.g. ML, Haskell), which make heavy use of pattern-matching, that the inclusion of overlapping patterns is a good way to reduce syntactic clutter and improve readability.

Predicates on pattern variables.

With the introduction of named variables on the LHS for input failures, it is tempting to abandon patterns altogether and instead use logical predicates (and, or, not, =, \neq) as guards on the RHS of the expression. For instance,

$$\begin{aligned} (f, g) \rightarrow \text{late}, & \quad \text{if } f = \text{late} \\ *, & \quad \text{if } f = \text{value and } g \neq \text{value} \\ \text{value}, & \quad \text{if } f = g = \text{value} \end{aligned}$$

We note that such predicates introduce considerable extra complexity in parsing the notation, with some detriment to readability, and no discernable difference in conciseness or expressibility. Thus, we have chosen to stick with patterns in FPTC (but compare and contrast with FPTN [2], see Section 6).

Full Syntax and Semantics of FPTC.

Given the design choices outlined above, we define the formal syntax of FPTC as in Figure 6. The outline evaluation algorithm of Figure 5 must also be refined to account for the full language features (see Figure 7). Because of the rule that exactly one clause should match any possible input permutation, it is necessary to ensure that at least one pattern always matches, but if more than one pattern matches, one must always be strictly more specific than the others. By defining the obvious partial-ordering — explicit faults and $*$ are more specific than variables and wildcards, but incomparable with each other — it is easy to detect and reject overlapping patterns (i.e. tuples with some elements greater and some lesser in corresponding positions). The same partial ordering also enables us to select the most specific matching clause where there is more than one. Any variables used in a LHS pattern are bound to actual faults and carried over to the RHS.

behaviour	=	expression+	
expression	=	tuple '→' tuple	
tuple	=	one	
		'(' one (' , ' one)+ ')'	
one	=	'*'	(no failure)
		'_'	(wildcard)
		alphachar	(variable)
		fault	(explicit fault)
		'{' fault (' , ' fault)+ '}'	(set of faults)
fault	=	'Value' 'Early' 'Late'	(user-extensible)
		'Omission' 'Commission' ...	

Figure 6. The EBNF syntax of FPTC.

$$\begin{aligned}
&\text{in}(c_i) = \text{out}(p_j) \\
&\quad \textbf{where } p_j = \text{predecessor}(c_i) \\
&\text{out}(c) = \bigcup_{tup \in \text{perms}(\text{in}(c_0), \dots, \text{in}(c_k))} \text{rhs}(\text{select}_{tup}(\text{transforms}(c))) \\
&\text{select}_{(f_0, \dots, f_k)} ts = t \in ts \bullet \text{lhs}(t) \text{ unifies with } (f_0, \dots, f_k) \\
&\text{lhs} \llbracket (g_0, \dots, g_k) \rightarrow (h_0, \dots, h_m) \rrbracket = (g_0, \dots, g_k) \\
&\text{rhs} \llbracket (g_0, \dots, g_k) \rightarrow (h_0, \dots, h_m) \rrbracket = (h_0, \dots, h_m)
\end{aligned}$$

Figure 7. Evaluation of the full FPTC language. Assuming a fixpoint calculation as in Figure 5, the dataflow equation for *in* sets is simpler, because each input connection has a unique source, rather than all inputs being aggregated together. But calculating *out* sets is more complex: inputs and outputs are tupled, so all permutations of inputs must be considered. For each permutation, exactly one FPTC clause must match, and is selected, yielding an output tuple.

5 Case Study

As a case study to validate the usefulness and compositionality of the failure propagation technique described here, we have applied it to LRAAM (Long-Range Air-to-Air Missile, also sometimes known as BVRAAM) [7,5] launch software, an avionics application specified in the Real-Time Networks notation.

In this study, the application already had a detailed architecture (see Figure 8), but the actual code was not available to us for precise analysis of its failure properties. Nevertheless, we were able to conjecture plausible transformation expressions for individual code units. Indeed, in many ways this is a good demonstrator for the utility of FPTC as a safety analysis prior to

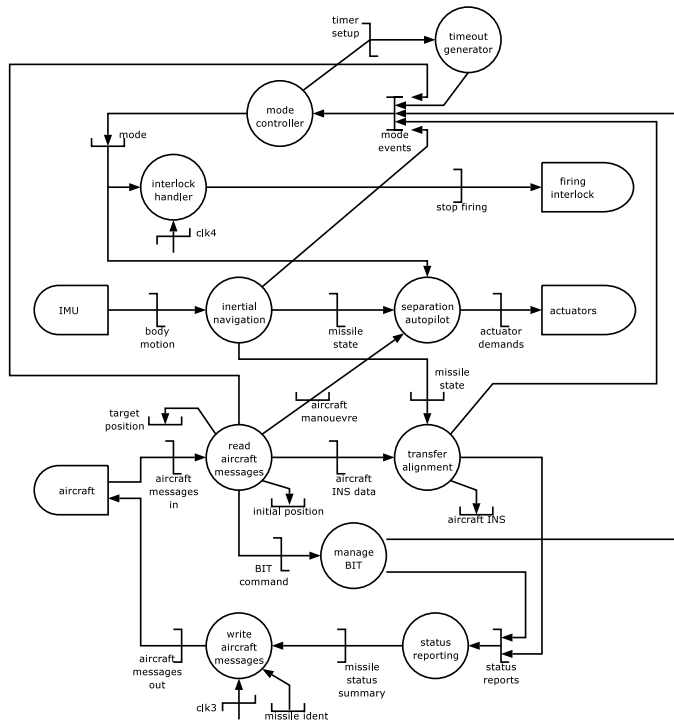


Figure 8. The LRAAM launch software architecture.

detailed implementation, because it can pick up potential problems before the code is written, when they are cheaper to fix. Another point worth noting is that the provided design is incomplete, specifying only the launch portion of the software, not the actual flight control (pools for communicating data to the flight phase are left unconnected). This too is realistic of real projects, where the full design is only performed after a demonstration of the viability of some critical portions.

The communications protocols between code units are a well-defined, well understood, reusable feature of the architectural style. They have been extensively studied in other contexts for their formal properties, and thus the failure expressions given for these connections (see Figure 4) are much more definite than those for the code units. Some examples of the FPTC expressions conjectured for the individual software threads are illustrated in Figure 9.

The essential structure of the LRAAM architecture is the central line of the diagram, which can be understood by comparison with the simple outline in Figure 1. The main sensor is an IMU (inertial measurement unit), feeding into an inertial navigation computation, which in turn feeds the separation autopilot, that drives the actuators (flight surfaces). The remainder of the

$$\begin{array}{l}
\text{mode controller} \\
\text{omission} \rightarrow (*, \text{detectable value}) \\
v \rightarrow (v, v) \\
\\
\text{inertial navigation} \\
v \rightarrow (v, v, v) \\
\\
\text{separation autopilot} \\
(v, w, x) \rightarrow \{v, w, x\} \\
\\
\text{transfer alignment} \\
(*, *) \rightarrow (\text{commission}, *, *) \\
(v, w) \rightarrow (\{v, w\}, \{v, w\}, \{v, w\})
\end{array}$$

Figure 9. FPTC expressions for some software threads in the LRAAM case study.

design is secondary, but deals with explicit mitigation of possible failures.

5.1 Results

We injected individual failure modes to potential sources of errors, e.g. the hardware sensor components, then performed the automated part of the analysis to determine how errors flowed through the network. Our tool annotates every arrow in the diagram with its final computed failure tokenset. For the purposes of this case study, we are especially interested in what failures might be expressed on the inputs to the flight control actuators.

The result of injecting a potential *omission* failure mode on the IMU sensor shows that the flight control hardware actuators might receive command data with a failure in the set $\{\text{late}, \text{stale value}\}$. Late actuation (or actuation with stale values) of the flight control surfaces might well be undesirable, but the result is hardly surprising given our failure assumptions on the sensors.

5.2 Change and Re-analysis

As already mentioned, one of the biggest costs of high-integrity software is associated with safety recertification after necessary change to the system, and our aim with this work is to enable a re-analysis to become simpler and cheaper. So to investigate the comparative ease of change in the present analysis, we make a small alteration to the architecture: by swapping the IMU sensor component for one with different failure properties. This is typical of the type of change one might make in the maintenance phase of a product, as hardware components become obsolete or unobtainable, and must be replaced.

With a different IMU failure mode, whereby the unit could instead provide

data too *early*, it turns out that the actuators now receive a failure in the singleton set $\{\textit{stale value}\}$. Thus we can see that, unlike the case of late sensor data leading to late actuation, the early sensor data does *not* lead to an early actuation. This type of failure has been repaired by the software architecture. However, the continuing presence of the *stale value* failure is initially rather puzzling. By manually examining the annotations of all the arrows backwards towards their sources, we quickly discover that the source of the stale value failure is actually the mode controller software unit: it seems that the channel which feeds events into the mode controller is susceptible to introducing a delay if too many events happen in rapid succession. One solution the designer could take is to derive a new safety requirement stating that events will not be generated faster than the mode controller clock can process them, and to prove this requirement is met by a careful timing analysis.

As a second example of a design change, we retained the original IMU hardware with a tendency to omit sensor readings, but decided to make it two-lane redundant, with a consolidator thread to combine the readings. Connections from IMUs to the consolidator were via the pool protocol rather than signals. The impact was that the actuators no longer have a *late* failure at all, but that the *stale value* failure can now be traced back not only to the mode controller, but also to the IMU consolidation unit.

5.3 Discussion

This type of change and re-analysis will typically be performed a large number of times throughout the lifecycle of the product. Other kinds of change that are easily handled include: the re-routing of some of the comms protocols to deliver data to different nodes in the network; the re-assignment of software processes to different CPUs; the addition of new software capabilities to the system, such as the completion of the LRAAM design to include flight control as well as launch control.

Every automated re-analysis of the LRAAM study by our tools took well under a second on a moderate performance PC. This reflects the essential simplicity of the symbolic processing required. The most intensive part of this technique is undoubtedly the required manual investigation of individual components for their failure transformation behaviours. Where the software components do not themselves exist yet, one can view the FPTC expressions as a specification against which to verify the real components at a later stage of development.

6 Related Work

There have been few results related to fault propagation in the literature – most analysis is instead focussed on proving correctness. Determining the safety of a critical system when it operates correctly is already a difficult problem, but preserving safe operation in the presence of faults is even more difficult.

The FPTC presented in this paper was influenced by the earlier Fault Propagation and Transformation Notation (FPTN) [2,3], one of an integrated set of methods for software safety analysis. The latter links a top-down Fault Tree Analysis (FTA) of software components with bottom-up Failure Modes, Effects, and Criticality Analysis (FMECA) of the same components. FTA starts from possible bad effects at the system level and works backwards to determine possible causes at the component level, whereas FMECA starts from known potential failure modes of components and pushes forwards towards their possible effects at the system level. FPTN is a bridge between the two methods. Essentially, it is a notation that permits analysis to progress either deductively (FTA) or inductively (FMECA), and to capture the information produced in a single place, allowing “end-to-end modelling of the failure behaviour of a system”[2].

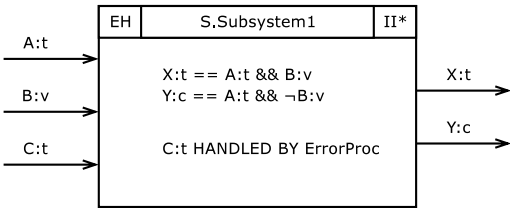


Figure 10. An example diagram in the original FPTN notation.

An example of an FPTN diagram is shown in Figure 10. The box is headed with the name of the component under study. To the top left and right are indications of (a) whether the component incorporates an exception handler (EH), and (b) the criticality level of the component (II*). Labelled arrows represent the direction of error flows, which might be a combination of dataflow, control flow, and other dependencies. (In a full system diagram, the arrows connect together different boxes representing the components.) The labelling of arrows is a pair consisting of a name, e.g. A, B, C (for reference in the transformation expressions), and a type, e.g. t, v, c, mnemonic for one of the HAZOP-style failure modes: *t*iming, *v*alue, *c*ommunications.

Transformation expressions in the body of the box are boolean equations relating the output failure modes to input failures. In this example, the output timing failure X is generated only in the presence of both the input timing fail-

ure A and an input value failure B. The timing failure C is handled internally to this component, and thus cannot propagate outwards.

We identify two key deficiencies in FPTN. First, whereas an FPTC analysis reuses the existing system design as the basis of its flow graph, the original FPTN is much more ad hoc, recording *only* known error flows. That is, an arc is included on a diagram only if it represents the transmission of a known particular fault type between components, but an actual data or control connection between components is *omitted* from the FPTN diagram if it is not currently known to cause error transmission. A later change to one component might alter not only the types of failure generated, but also where those failures are propagated to, thus necessitating some non-local re-working of the FPTN diagram to change the connection topology. The only way to determine the new connection graph is by manual reanalysis of all the components and their contexts. By contrast, FPTC diagrams are based on the full architecture used for developing the software code itself, to identify and record all potentially important dependencies, whether or not they are currently known to engage in any error flow. Thus FPTC keeps the model and reality synchronised as much as possible, and localises the effect of any changes.

Secondly, the development of transformation expressions for components in FPTN is not modular. Each component is analysed only within its system context. Incoming labels identify only the input failures that definitely can arise from other components, and output labels are based solely on the given input failures and potential internal failures. Thus, any one FPTN expression is fragile to minor changes in *other* components. Our FPTC is more robust, in requiring that each component be analysed in isolation for all possible failure responses, not just those in the currently-known context.

As a result of these two deficiencies, the analysis of failure propagation cannot be automated within FPTN. It remains strictly a notation for recording existing human-derived knowledge, with no useful algebraic or predictive properties.

Grunske [4] demonstrates a hierarchical version of FPTN, where the failure expression boxes are nested. Whilst this provides a useful generalisation of the FPTN technique, it suffers from the same problems as the original FPTN. Indeed, the example given there is an exceptionally clear demonstration of the lack of predictive connection between the system architecture and any consequent failure analysis.

7 Conclusions and Future Work

We have presented a new analysis for determining the failure behaviour of a HIRTS system of software components based solely on information about the failure behaviour of the components themselves, and their connections. The system-level analysis is compositional and automated, unlike any previous fault analysis. However it does rely on the user correctly identifying all the potential failure modes of interest, and developing accurate models of the consequent failure transformation behaviours of individual components.

The technique has been used in practice on a case study of an avionics application, where the design was incomplete, and the implementation unavailable. This demonstrates the utility of the analysis at the early stages of a project, in order to avoid a potentially hazardous design. We also demonstrated the ease of re-analysis in the face of design changes of different types. This leads us to believe that the technique could have particular value in the fields of incremental safety certification, and product-line families.

The selection of an unambiguous notation for expressing failure transformations is a subtle problem, and we have made several simplifying assumptions in order to keep the notation tractable for human readers.

Goals for the future include the extension of our FPTC framework to look at Quality of Service issues. If we re-designate the symbolic tokens of FPTC from failure modes to representations of the quality of system data, that is, our confidence in its accuracy, (e.g. Low, Medium, High), or the severity of any associated failure, then it may be possible to notate the transformational efficacy of common components such as multi-lane voters, redundant wiring schemes, and so on. Do value faults get amplified through feedback loops, or reduced? Is low quality data relied on in high-integrity partitions? This could enable us to be more confident about the impact of change in a large and complex design, detecting whether the addition or removal of elements introduces a possible single point of failure.

Reliability modelling could be achieved by attaching probabilities to the various possible outcomes of propagation rules. By injecting known reliability data for the hardware components, one can then compute predicted failure rates for the system as a whole.

Another area we are currently investigating is whether a *tracing* semantics can be attached to error tokens in the standard FPTC. At the moment, if it be surprising to the user that a particular failure mode has been calculated to exist on a dependency path, then some manual investigation and reasoning is needed to discover the true source of the failure mode elsewhere in the system, through its various possible propagation paths and transformations

of type along the way. If, instead, each error token were annotated with the entire path it has taken through the graph, it would provide an instant ‘explanation’ for the user. However, we need to be careful on the one hand that all possible traces are gathered, yet on the other hand that cyclic traces are ‘short-circuited’ to ensure that the decision algorithm terminates.

Other areas for future work are:

- Is it possible to generate safety case data for certification authorities, directly from the FPTC analysis?
- How do we verify that the implementation is “no worse than” the FPTC says, and can we prune failure conditions if it is “better than”?
- Can we automate the determination of an FPTC expression for individual components, say by symbolic execution of the actual implementation (e.g. coded in SPARK Ada) in the presence of simulated faults?

References

- [1] Andrew W Appel. *Modern Compiler Implementation in C*, chapter 11: Liveness Analysis. Cambridge University Press, Cambridge, UK, 1998.
- [2] Peter Fenelon and John A McDermid. New directions in software safety: Causal modelling as an aid to integration. Technical report, High Integrity Systems Engineering Group, Dept of Computer Science, University of York, 1992.
- [3] Peter Fenelon and John A. McDermid. An integrated toolset for software safety analysis. *The Journal of Systems and Software*, 21(3):279–290, June 1993.
- [4] Lars Grunske and Roland Neumann. Quality improvement by integrating non-functional properties in software architecture specification. In *EASY’02: Second Workshop on Evaluating and Architecting System Dependability*, pages 23–32, San Jose, California, October 2002.
- [5] Peter A Lindsay and John A. McDermid. Derivation of safety requirements for an embedded control system. In *Systems Engineering Test and Evaluation Conference*, pages 83–93. Systems Engineering Society of Australia, 2002.
- [6] J A McDermid, M Nicholson, D J Pumfrey, and P Fenelon. Experience with the application of HAZOP to computer-based systems. In *Compass ’95: 10th Annual Conference on Computer Assurance*, pages 37–48, Gaithersburg, Maryland, 1995. National Institute of Standards and Technology.
- [7] S E Paynter. A BVRAAM case study in safety engineering: Specification and design. Technical Report DR 23121, MBDA Missile Systems, December 2001.
- [8] S E Paynter, J A Armstrong, and J Haveman. ADL: An activity description language for real-time networks. *Formal Aspects of Computing*, 12(2):120–140, Feb 2000.
- [9] H R Simpson. The MASCOT method. *Software Engineering Journal*, 1(3):103–120, March 1986.