Capture and reuse of composable failure patterns

Ian Wolforth*, Martin Walker and Yiannis Papadopoulos

Department of Computer Science,
University of Hull,
Hull, HU6 7RX, UK
E-mail: i.p.wolforth@dcs.hull.ac.uk
E-mail: martin.walker@hull.ac.uk
E-mail: y.i.papadopoulos@hull.ac.uk
*Corresponding author

Lars Grunske

Faculty of ICT,
Swinburne University of Technology,
John Street, Hawthorn, Melbourne,
Victoria 3122, Australia
E-mail: lgrunske@swin.edu.au

Abstract: Emerging safety analysis techniques use composition of failure models or fault simulation in formal models of a system to determine relationships between the causes and effects of failure. Most recent work has focused on developing system modelling and algorithms for automatic safety analysis. However, little work has focused on developing principles to improve reuse of safety analyses in the context of these techniques. In this paper, we describe a generalised failure logic (GFL) that can capture abstract reusable characteristics of failure behaviour and show how the GFL can be used with templates for the specification of reusable and inheritable component failure patterns. Finally, we illustrate how such patterns can be used with Hip-HOPS, an automated fault tree and FMEA synthesis tool, in order to simplify safety analysis while formalising and improving reuse. Benefits of this approach are discussed in the light of a case study on a brake-by-wire example.

Keywords: safety patterns; reuse in safety analysis; automated FTA; automated FMEA.


Biographical notes: Ian Wolforth is a PhD student at the Department of Computer Science, University of Hull. He obtained his BSc in Software Engineering and MSc at the University of Hull. The title of his MSc thesis was ‘Combining fault tree synthesis with efficient evaluation of fault trees’. His PhD topic focuses on reuse within safety analysis, in particular patterns of component failure behaviour. He has published work in the Proceedings of the Third International Conference on Dependability of Computer Systems, DepCoS-RELCOMEX’08.
1 Introduction

Safety critical systems are systems which have the potential to endanger people or the environment when they fail. As a result, a considerable body of work has been developed to ensure that potentially hazardous failures are mitigated or prevented as far as possible. Key steps include identification of system failures, establishment of their severity, identification of possible causes and calculation of probability of occurrence, in a process known as safety analysis. By understanding the relationship between system failures and their causes, the effects and likelihood of failures can be minimised by preventing those causes or modifying the system design to make them less likely.

There are many different safety analysis techniques; two of the foremost are fault tree analysis (FTA) (IEC 61025, 1990; Vesely et al., 2002), in which the combinations of possible causes are deduced from the system failure, and failure modes and effects analysis (FMEA) (IEC 60812, 1991), which analyses the possible effects each failure can have on the system. These techniques have traditionally been carried out manually, involving a team of engineers labouring to produce comprehensive safety documents in order to fulfill a safety specification. A great deal of knowledge about the failure behaviour of the system is gained during this process, but due to the time and expense involved, such a study is typically only carried out once, after the design is complete, in order to determine whether the design meets its safety goals. In addition, as systems grow more complex, involving multiple states or phases of operation and more intricate programmable components, a manual analysis of the system design is both increasingly difficult and potentially more error-prone, resulting in an imperfect or incomplete view of
the system failure behaviour. It would be more desirable to be able to analyse the system more than once, feeding the results back into a new design as part of an iterative process; this is potentially more cost-effective – since major design flaws can be identified earlier, obviating the need for costly changes at a late stage – and also results in a safer, more reliable system.

1.1 Automated safety analysis: basic concepts and the issue of reuse

One way of simplifying the process and of enabling useful iterations of safety assessment is to automate (or partly automate) the analysis process. In industry, there are established software tools that automate calculations on manually constructed fault trees or assist clerical tasks in essentially manual FMEA processes. However, the synthesis of predictive system failure models such as fault trees and FMEAs remains manual. Over the last 15 years, research has focused on further simplifying safety assessment by automating the synthesis process. This work has followed two different paradigms, each defining a distinct way of synthesising system failure models from other system information. The first paradigm can be called ‘compositional failure analysis’ while the second ‘behavioural fault simulation’ (Heitmeyer et al., 1998; Reese and Leveson, 1997). In compositional failure analysis, system failure models are constructed from component failure models using a process of composition. System failure models are, or can be automatically translated to, well-known dependability evaluation models including fault trees, stochastic Petri nets and Markov chains. Techniques that follow the compositional approach include: Failure Propagation and Transformation Notation (Fenelon and McDermid, 1993), HiP-HOPS (Papadopoulos and McDermid, 1999), Component Fault Trees (Kaiser et al., 2003; Grunske and Kaiser, 2005), State-Event Fault Trees (Grunske et al., 2006) and Failure Propagation and Transformation Calculus (Wallace, 2005). On the other hand, in behavioural fault simulation, system failure models equivalent to an FMEA are produced by injecting faults into executable formal or semi-formal specifications of a system, thereby establishing the system-level effects of faults. Techniques that follow this approach include safety analysis using Altarica (Bieber et al., 2002), FSAP-NuSMV (Bozzano et al., 2003), software deviation analysis (Heimdahl et al., 2005) and DCCA (Güdemann et al., 2007).

Individual techniques have their strengths and benefits. Behavioural simulation can, in theory, achieve a higher degree of automation because a tool like a simulator or a model-checker is expected to do most of the assessment. However, the effort required for system modelling, especially formal system modelling, should not be underestimated. Higher automation comes at a higher computational cost than compositional safety analysis techniques, which typically employ algorithms of lower complexity. Most behavioural simulation techniques are also inductive, i.e., the assessment proceeds from known causes to unknown effects, and in this type of analysis effective assessment of combinations of causes is very difficult – if not impossible – to achieve due to combinatorial explosion.

A detailed comparison of these techniques is beyond the scope of this paper [the reader is referred to the work of Lisagor et al. (2006)]. However, it is important to point out that most work has focused on system modelling and automation of the analysis, while very little work has been done on developing principles that will enable or maximise reuse of component or system safety analyses across applications. A notable exception to this is the work by Kelly and McDermid (1997) on specification of safety
Capture and reuse of composable failure patterns

In this paper, we examine the problem of reuse in safety analysis and develop linguistic concepts that extend Hierarchically-Performed Hazard Origin and Propagation Studies (HiP-HOPS), an established compositional safety analysis technique, to enable such reuse via specification and use of component failure patterns.

1.2 HiP-HOPS: basic concepts and motivation for reuse

HiP-HOPS takes as input a topological model of the system (e.g., as modelled in a CAD tool) that has been annotated with component failure data, describing how each component fails internally and how it reacts to failures elsewhere and performs both FTA and a multiple failure mode FMEA on this model. The analysis of propagation of failures is deductive (from effects to causes) and therefore the technique is not prone to combinatorial explosion. This has enabled not only application of the technique to large systems but also its combination with greedy heuristics for the purpose of architectural optimisation with respect to dependability and cost (Papadopoulos and Grante, 2005).

HiP-HOPS offers a semi-automated process that enables a designer to identify weaknesses in the system and therefore improve the design to remedy them. However, it still requires the analyst to annotate each system model and this requires additional time and effort. If it were possible to reuse data from earlier analyses, the process could be made even more efficient. Unfortunately this is often difficult, as the failure data annotations are specific to each system and even each component. It is possible to partially overcome this problem by employing reusable components in the design and storing these annotated components in a library, so that subsequent system designs can use previously annotated components, but the problem remains that the failure data is often too specific to be easily reusable.

To avoid this problem, system design and software engineering disciplines use the notion of a pattern of behaviour – a generic template of behaviour that can apply to multiple components or other design entities (Pullum, 2001). However, transferring these ideas to safety analysis has had only limited success thus far. In HiP-HOPS and most other emerging safety analysis techniques, specifications of how components fail are too context specific, e.g., they make references to specific inputs received and outputs produced by the component and the deviations from intended behaviour to these parameters. Such models cannot be easily reused in different contexts of application. In HiP-HOPS, for example, component-level failure specifications are currently formed as sets of logical expressions that describe specific failure behaviour by relating specific output deviations to internal malfunctions of each component and deviations of component inputs.

In this paper, the concept of a local, component-level failure specification is extended to enable description and reuse of more generalised failure behaviour in the form of patterns. The paper is organised as follows: in Section 2, we show that components do exhibit patterns of failure worth capturing, storing and re-using in safety analysis. We also discuss the current language for describing failure behaviour in HiP-HOPS and identify its limitations. In Section 3, we present an extension of that language in the form of a generalised failure logic (GFL) that enables designers to represent patterns of failure behaviour. Patterns described in GFL could in practice be used to capture common types of fault propagation, fail silence and fault tolerance that components are designed to
Patterns of failure behaviour

2.1 Reusing failure patterns

Enabling a greater degree of reuse in safety analysis is potentially very beneficial. Not only does it save time and effort, but the experience and knowledge gained in previous analyses can be drawn upon in subsequent analyses too. This is especially important if, for example, the first analysis was by a different analyst who perhaps had a greater degree of understanding of the failure behaviour of those components. In this way, reusability also provides a mechanism for providing a continuity or flow of knowledge from one system design to the next. The end result is a more robust design containing common components with well-understood behaviour.

Patterns are one way of accommodating this sort of reusability. By examining many different components and their failure behaviour, it is possible to see certain types of behaviour recurring – effectively, common failure behaviours shared by many different components. For example, one of the simplest patterns of failure is the notion of a ‘propagator’ – a component that does not react to failures at its input except to propagate them to its output. Many connectors in systems are types of propagators; for example, both an electrical connection and a hydraulic hose will propagate an omission of input (no current, no flow) to their outputs, both will propagate a low value (e.g., low voltage, low pressure) to their outputs and both will propagate an unexpected input (e.g., a voltage spike caused by a short circuit, sudden flow caused by a faulty valve, etc.) to their outputs. This pattern of behaviour could be generalised by saying that a propagator will propagate any type of failure (known as the failure class) at its input to a corresponding failure at its output. This ‘propagator’ behaviour could then be assigned to many different types of connector and reused with the knowledge that an input failure of any failure class will cause an output failure of the same class, regardless of the type of connection or the nature of whatever is carried along the connector. In reality, of course, components that exhibit ‘propagator’ behaviour would still fail internally in unique and different ways and somehow such failures would also need to be captured in any specification of failure. The abstract ‘idea’ of a perfect propagator – however imperfect the concept is in reality – can still be useful as an aide for understanding.

Generalisation is not only possible in terms of the failure class: it is also possible to generalise the inputs and outputs. For example, a communications bus is typically a form of propagator: it serves only to communicate data at an input to an output. If there is no input, there is no output, and if there is an unexpected input, there is an unexpected output (the provision failure class, i.e., omissions and commissions); if there is corrupt data at an input, there will be corrupt data at the output (the value failure class); if the data at an input is early or late, then the data at the output will similarly be early or late (both timing failures). However, a bus does not necessarily have just one input and one output. It may have many inputs and many outputs, depending on how many components are connected to the bus; therefore, it is necessary to generalise the number and nature of
the inputs and outputs as well. It would then be possible to say that a ‘propagator’ will propagate any input failure to all outputs, or alternatively, that any output failure is caused by the same class of failure at any input.

Not all patterns are so simple. Another possible pattern of failure – or rather, an attempt to mitigate failure – is the standby-recovery pattern, where a primary component is replicated to provide a standby that will take over its function if the primary fails. In this instance, an omission of output from the primary and any failure from the standby is required for the subsystem to fail. However, it is also possible for a value failure to go undetected by the primary and in this case the standby has no effect. The standby-recovery pattern, then, is more complicated than the simple propagator because of the fact that different failure classes can yield different failure behaviour. Furthermore, the standby-recovery pattern is itself a form of the redundancy pattern, where components are duplicated and all must fail to cause the system to fail.

As another example, consider the TTP/C communication controller (Kopetz and Grünsteidl, 1994) shown in Figure 1 – a component that handles communication between a host and other controllers in a time-triggered network. Like many complex components, particularly those designed with fault tolerance in mind, the TTP/C controller is designed with the **fail-silent** pattern: in the event of a failure, it will lead to an omission of output from the controller. In a time-triggered network where messages are sent using a predefined schedule, omission of messages by the sender is detectable by all receivers and, therefore, more preferable to other, more hazardous failure types, such as the commission, timing or value failures.

![Figure 1](image.png)

The TTP/C controller has several important safety properties, designed to hold in any operational context: timing or commission failures caused by internal faults or received at the input (e.g., from the host) can be detected and will result in an omission of output, meaning that these failure classes are not propagated to other components in the system. In effect, commission or timing failures are transformed into omission failures, which are deemed less severe as they are detectable by all receivers. This means that the designer only needs to consider how to deal with omission and undetectable value failures in the TTP/C network.

However, at present, such patterns can only be used in an ad hoc and informal way; before they can be applied in a more automated fashion, or reused in a more flexible manner, there needs to be a more formalised method of representing the patterns – or in
the case of a technique such as HiP-HOPS, the existing representation must be extended to accommodate the pattern semantics.

2.2 HiP-HOPS failure logic

In HiP-HOPS, model-based synthesis and analysis of fault trees and FMEA is fully automatic. However, before such analysis can take place, each component in the system model must first be annotated with local failure logic. This logic describes how each component fails by relating failures at its output (known as output deviations, i.e., deviations from normal operation) to a logical combination of corresponding input deviations and internal failures of that component.

Each component is therefore annotated with a number of logical expressions of the following form:

\[
\text{Output Deviation} = \text{Internal Failures AND/OR Input Deviations}
\]

The output and input deviations consist of two parts separated by a dash. The first part is the failure class of the deviation, typically abbreviated; for example, O normally indicates an omission failure, C a commission; V may be a value failure, LV and HV might be low and high value, etc. The second part is the name of the port where the deviation occurs. HiP-HOPS uses ports as a simple abstraction to represent the interface between a component and a connection (or a component and another component). Typically, there will be one port for each input and each output in a component and these are then named so that they can be referred to in the expression. Thus ‘O-in1’ would be an omission of input at port ‘in1’ (assuming that ‘in1’ is an input port) and ‘C-out1’ would be a commission of output at port ‘out1’. Deviations can also contain parameters, so that they can refer to properties of a port as well; for example, ‘HV-fluidIn-pressure’ would be a value failure (HV = High Value) of parameter ‘pressure’ at port ‘fluidIn’.

Internal failures correspond approximately to basic events in FTA or failure modes in FMEA: these serve as the root causes of any system or component failure, whilst the deviations serve as a mechanism to show the propagation of those failures through the system. Internal failures are often given additional data, such as failure and repair rates; these values can then be used to perform a probabilistic or quantitative analysis of the system. Input deviations and internal failures can be mixed in any combination by means of the standard Boolean logical operators AND and OR and this forms the logical part of the expression; there is one expression per output deviation in HiP-HOPS.

This syntax is relatively simple and quite flexible, but it struggles to efficiently represent patterns of behaviour such as those mentioned above. For example, in the case of the propagator pattern, HiP-HOPS would require one expression per failure class:

\[
\begin{align*}
O-\text{out} &= O-\text{in} \\
C-\text{out} &= C-\text{in} \\
V-\text{out} &= V-\text{in} \\
\end{align*}
\]

Similarly, HiP-HOPS also requires one expression per output deviation, so in the case of a bus with three parameters and only one input and one output, three expressions would be needed just to represent each failure class:
Capture and reuse of composable failure patterns

O-out1-param1 = O-in1-param1
O-out1-param2 = O-in1-param2
O-out1-param3 = O-in1-param3

C-out1-param1 = C-in1-param1

This is clearly not the most efficient approach. The problem can also be seen in practice by attempting to annotate the fail-silent behaviour of the TTP/C controller:

O-out = O-in-p1 OR O-in-p2 OR O-in-p3 OR…
OR C-in-p1 OR C-in-p2 OR C-in-p3 OR…
OR Early-in-p1 OR…
OR Late-in-p1 OR…
V-out = V-in-p1 OR V-in-p2 OR V-in-p3…

This type of annotation is arduous to carry out even once and is far from reusable: if a component was substituted with an alternative version that had the same failure behaviour but additional ports, then it would require new expressions to be added at best, and at worst, all expressions would need rewriting to account for the extra inputs. It would be far better if it was possible to capture this behaviour with just one or two generalised expressions instead.

3 Generalised failure logic

3.1 Generic expressions

Components with more complex functions often require application-specific terms in their HiP-HOPS annotations to successfully describe their failure behaviour. This can make it difficult to reuse such annotations in other scenarios where changes to the operation or interface of the component make them incompatible with the new system. This means the annotations must be reconstructed to better reflect the component’s new context – a manual process which can be time-consuming and can potentially introduce errors to the expressions.

However, it is possible to make the observation that although the terms of the expressions often change, whether in number or in the failure classes or parameters used, the logical structure of the expression will often stay the same. For example, a component with the ‘propagator’ failure behaviour – such as a bus – will propagate any type of failure from input to output. The current method of describing this would require a set of expressions, one for each failure class:

O-out = O-in
C-out = C-in
V-out = V-in
It is only the failure class that changes with each expression, whilst the logical structure of the expression (in this case, a simple propagation) remains the same. Therefore, a more abstract form of the expression that represents the propagation behaviour in a generalised format could provide a more reusable annotation, one that can be applied independently of system context.

The GFL is designed to achieve just that. It introduces abstractions into the HiP-HOPS expressions to produce generalised failure expressions (GFEs), which are more generic descriptions of the component failure behaviour. As a result, it is often possible to annotate a component with one or two GFEs where it would take several non-generic HiP-HOPS expressions to achieve the same thing. The two primary syntactic features that allow this generalisation are vectors and operators.

Using the above bus annotations as an example, if the changing failure classes are substituted for a single abstract ‘vector’ of possible failure classes, it is possible to create a general form of the expression that has the same reusability properties as the set of simple expressions shown earlier:

\[ \text{[failure class]} \text{-out} = \text{[failure class]} \text{-in} \]

The vector ‘FC’ is used to perform this function: an abstract term that can be substituted to represent any appropriate failure class in a GFE. FC is the vector of all possible failure classes in a system model. To be able to make further generalisations, there are vectors to represent ports – ‘OP’ for all output ports in a component and ‘IP’ for all input ports – as well as for parameters, where ‘PM’ represents all relevant parameters for a given port. It is also possible to define subsets of a vector containing only specific items by means of a list, e.g.:

\[ FC : \{O, C\} \text{-out} = \ldots \]

is a list of failure classes containing only O (omission) and C (commission). It is also possible to define exceptions to a vector, meaning that all members of the vector are used except for those specifically mentioned, e.g., to define a set of all failure classes except value failures:

\[ FC \text{ EXCEPT } \{V\} \text{-out} = \ldots \]

The left-hand side of a GFE is known as a generalised output deviation (GOD). In this context, vectors indicate that the expression applies to every member of the vector; effectively, a GOD represents multiple standard HiP-HOPS expressions. It is also possible to use vectors on the right-hand side to create one or more generalised input deviations (GIDs), but to do so, it is first necessary to introduce the concept of operators.

In the ‘propagator’ pattern, the propagation of failures is defined such that a failure at the output of a component is caused by the same failure at the input of that component. In this case, it is necessary to define a correspondence between the input and output sides of the GFE and that can be achieved by means of the SAME operator:

\[ FC \text{-out} = \text{SAME}(FC) \text{-in} \]

Without the SAME operator, the expression would imply that FC can have any value on either side of the expression. Therefore, the SAME operator is necessary to capture the propagation behaviour of the bus in this GFE: the bus will propagate any class of failure,
so by substituting the same valid failure class for the abstract vectors FC on both sides, it is possible to obtain the specific failure behaviour for that failure class. In this way, the designer can represent the pattern of failure behaviour directly within a GFE, instead of attempting to capture it with multiple non-generic expressions.

But what of combinations of input deviations? For example, one component in a system model may be designed to omit output in response to an omission failure at any of its inputs. This can be described with a GFE as follows:

\[ \text{O-out} = \text{O-ANY(IP)} \]

The ANY operator defines a logical disjunction of all input ports (represented by the IP vector). This single GFE would only represent a single non-generic HiP-HOPS expression, but the expression would have to contain multiple input deviations, one for each of the \( n \) input ports, like so:

\[ \text{O-out} = \text{O-in1 OR O-in2 OR … O-in} \]

The GFE equivalent, by contrast, is reusable for any number of input ports. If the component is altered or substituted with another that uses a different number of inputs, the failure behaviour – as represented by this GFE – still holds and does not require updating. The use of operators such as ANY, which make collective references to structural or semantic properties of the system model, can greatly improve the level of generalisation possible using the GFL. For instance, it is possible to modify the previous example component to be more robust, such that it omits output only in response to an omission of all inputs. This can be achieved by means of the ALL operator, which applies a logical conjunction to a vector:

\[ \text{O-out} = \text{O-ALL(IP)} \]

Note that GFEs can also include non-generalised elements, e.g., ‘O-ALL(IP) OR InternalFailure’; in this case, the standard Boolean operators AND and OR can be used to connect basic events and GIDs as well as standard input deviations.

By using the vectors and operators, the GFL can capture many different patterns of failure behaviour as sets of GFEs, including more complex behaviour than simple logical disjunctions or conjunctions. The ‘voter’ pattern is used to describe the failure behaviour of components that are designed to be fault tolerant; a voter component will only propagate a failure if that failure occurs at a majority of its inputs. It is also known as the \( k\text{-out of-}n \) pattern, where \( k \) represents the majority of \( n \) (two out of three; three out of four; three out of five; etc.). Frequently, this will be applied to a redundant subsystem containing \( n \) components, where \( k \) of the \( n \) components must fail to cause a failure of the entire subsystem. This ‘voter’ pattern can be easily represented using the MAJ (majority) operator:

\[ \text{O-out} = \text{O-MAJ(IP)} \]

Assuming \( n \) (the size of IP) is 3, this GFE is equivalent to:

\[ \text{O-out} = (\text{O-in1 AND O-in2}) \text{ OR } (\text{O-in1 AND O-in3}) \text{ OR } (\text{O-in2 AND O-in3}) \]
This is a powerful generalisation: not only does it significantly reduce the size of the expression, but it is also reusable on any scale; using traditional syntax, the number of input deviations required is \( n(n-1) \), whereas only a single term is required in a GFE.

However, a GFE by itself is not sufficient to provide an analysis with meaningful data about specific failures – it is only an abstraction. The utility of GFEs in practice therefore relies upon the twin concepts of instantiation and expansion: the means by which concrete descriptions of failure behaviour are obtained from their generalised representations. This requires the context – the number of ports, the possible failure classes, the relevant parameters, etc. – which can be automatically obtained from the system model, with additional information provided by the designer (in a template – see below). It can also be expressed directly with lists or exceptions.

Instantiation is applied to the GODs – the left-hand side of the GFEs. A single GOD can represent multiple normal output deviations and each of these would be required for analysis. To obtain the normal output deviations, all the vectors in the GOD are enumerated and a new expression created for each combination of items. For example, given \( \text{OP} = \{\text{out1, out2}\} \) and \( \text{FC} = \{\text{O, C}\} \):

\[
\begin{align*}
\text{FC-OP} = \ldots & \rightarrow \text{O-out1} = \ldots \\
& \quad \text{O-out2} = \ldots \\
& \quad \text{C-out1} = \ldots \\
& \quad \text{C-out2} = \ldots 
\end{align*}
\]

Thus in this case, the GOD is instantiated into four separate expressions.

Before these expressions can be used in analysis, however, any GIDs must first be expanded. Whereas generalisation in an output deviation represents multiple expressions, generalisation in an input deviation represents multiple terms of the same expression, as demonstrated above by the \( \text{MAJ} \) operator for instance. Because vectors only occur in a GID in collaboration with an operator, to expand the vector, the operator is simply applied to every term within the vector, e.g., assuming \( \text{IP} = \{\text{in1, in2, in3}\} \):

\[
\begin{align*}
\text{O-ALL (IP)} & \rightarrow \text{O-in1 AND O-in2 AND O-in3}
\end{align*}
\]

Once instantiated and expanded like this, a GFE can provide the required set of local failure expressions required to perform analysis. Note that GFEs can also include non-generalised input elements, e.g., ‘\( \text{O-ANY}(\text{IP}) \) OR \text{InternalFailure} \)’, and these do not need expanding since all GIDs and non-generalised inputs are connected by normal Boolean operators (OR, AND).

### 3.2 Pattern templates

However, there is still a wider issue. GFL provides the syntactical capability required to produce generalised expressions that can represent patterns of failure behaviour, but it cannot guarantee that the resulting generalised expressions are reused appropriately. Often, reuse of component annotations is performed in an ad-hoc ‘cut and paste’ manner, which is potentially dangerous since mistakes are easily introduced, potentially leading to hazardous behaviour going unidentified in the model or non-existent.
failure behaviour being incorrectly represented. To avoid issues of invalid or incorrect application of GFEs, there must be some mechanism for the effective management of their reuse. To fulfil this purpose, we propose the use of pattern templates, a form of documentation similar in purpose to the design pattern documentation used in software engineering.

It is essential to have a consistent (and machine-readable) format for the documentation of safety patterns in order to promote safe and appropriate reuse. By constructing and reading a component’s reusability template, it is easier for the designer to understand when and where the component can be safely reused. An example template for a simple propagating bus is below:

<table>
<thead>
<tr>
<th>PATTERN TEMPLATE:</th>
<th>Propagator bus</th>
</tr>
</thead>
<tbody>
<tr>
<td>DESCRIPTION:</td>
<td>Propagates any failure at any input to every output.</td>
</tr>
<tr>
<td>USAGE NOTES:</td>
<td>Designed for use with normal failure classes such as omission, value, commission, etc., and data-based input/output parameters. Can work with any number of input/output ports.</td>
</tr>
<tr>
<td>FAILURE LOGIC:</td>
<td>ANY(FC)-OP = SAME(FC)-ANY(IP)</td>
</tr>
<tr>
<td>INSTANTIATED EXAMPLES:</td>
<td>O-out-signal = O-in-signal</td>
</tr>
<tr>
<td></td>
<td>V-out-signal = V-in-signal</td>
</tr>
<tr>
<td></td>
<td>etc.</td>
</tr>
<tr>
<td>RELATED TEMPLATES:</td>
<td>Specialised bus, multiplexer, demultiplexer</td>
</tr>
</tbody>
</table>

This template contains the elements needed to ensure correct reuse. It begins with a name, which can be used to uniquely identify the template, and then a text description of what it does, followed by some guidance on where it should and should not be reused together with some instructions on which parameters and failure classes are applicable. Next is the actual failure behaviour, represented by GFEs; the information required to instantiate these expressions can normally be obtained directly from the model. Finally, there are some illustrative examples in standard HiP-HOPS format to show how the GFEs might be instantiated and optionally a section listing related templates that may also be useful.

Reusability templates like this have another benefit: they make it possible to create a specialisation of a pattern by deriving one pattern from a base pattern and incorporating additional, more specific information. In the GFL, this is accomplished by means of the ‘INHERITS’ directive – so called because the derived template will ‘inherit’ the GFEs of the original. Specialisation is achieved by adding new GFEs which will override inherited behaviour.

For example, we may have a fail-silent pattern template with the following GFE:

<table>
<thead>
<tr>
<th>PATTERN TEMPLATE:</th>
<th>Fail-silent</th>
</tr>
</thead>
<tbody>
<tr>
<td>FAILURE LOGIC:</td>
<td>Omission-OP = ANY(FC)-ANY(IP)</td>
</tr>
</tbody>
</table>

This states that an omission of all output ports will be caused by any failure class occurring at any input port. We may then decide to use a TTP/C controller in a system model, but rather than creating an entirely new template for the controller we can inherit
from this basic fail-silent pattern. However, as mentioned earlier, the TTP/C controller cannot detect value failures: these will be propagated. Therefore, this behaviour must be overridden:

<table>
<thead>
<tr>
<th>PATTERN TEMPLATE</th>
<th>TTP/C controller</th>
</tr>
</thead>
<tbody>
<tr>
<td>INHERITS</td>
<td>Fail-silent</td>
</tr>
<tr>
<td>FAILURE LOGIC</td>
<td>Value-OP = Value-ANY(IP)</td>
</tr>
</tbody>
</table>

This will ensure that, for any input failure of class ‘value’, a corresponding output value failure will occur; however, for all other failure classes, the behaviour inherited from fail-silent will apply instead.

As an example, the propagator bus template presented earlier in this subsection can be inherited from to create a ‘specialised bus’, which includes the standard propagation behaviour from the base template but adds a new expression to reflect the fact that the ‘specialised bus’ can also generate internal failures of its own:

<table>
<thead>
<tr>
<th>PATTERN TEMPLATE</th>
<th>Specialised bus</th>
</tr>
</thead>
<tbody>
<tr>
<td>INHERITS</td>
<td>Propagator bus</td>
</tr>
<tr>
<td>FAILURE LOGIC</td>
<td>V-OP = V-ANY(IP) OR InternalFailure</td>
</tr>
</tbody>
</table>

In this case, value failures are caused either by input value failures or an internal failure of the bus; in the base pattern, value failures are only caused by corresponding value failures at the input. Hence, in the case of value output deviations, the new behaviour is used, but in all other cases, the inherited behaviour is used.

Templates and specialisation provide a mechanism which keeps components and their behaviours together while also providing the scope to create and reuse documented, well-known patterns. Thus, with suitable pattern documentation, it is possible to minimise inappropriate reuse, potentially eliminating unmanageable ad-hoc approaches from the process of component annotation.

4 Case study

4.1 Brake-by-wire system

To see how this generalised failure language can be applied in practice, shown in Figure 2 is a simple model of a distributed car braking system. It consists of a brake pedal that is connected to the brakes at the wheels by means of a redundant pair of buses and a number of TTP/C controllers.

When pressed, the brake pedal transmits a signal to a TTP/C sender, which sends that signal via two buses. The system is configured such that it can still operate even if one bus fails, providing a degree of redundancy. For each wheel brake, there are two TTP/C controllers, one connected to each bus, to receive the signal and translate this into a command to apply braking pressure. Again, these are replicated for redundancy: a brake can still function if one of the two TTP/Cs fails.
Figure 2  Distributed braking system
These components have been modelled in the modelling tool MATLAB/Simulink and then annotated with additional failure behaviour, specified in the GFL. The local failure data for each component is given below, but note that these expressions have been simplified by assuming there is only one parameter in each port – the braking signal – meaning abstract references to ‘any’ or ‘all’ parameters are unnecessary:

**Pedal**
- Omission-out = PedalFailed
- ValHigh-out = PedalBiasedHigh
- ValLow-out = PedalBiasedLow

**TTP/C Sender**
- Omission-out = ANY(FC) EXCEPT {ValHigh, ValLow}-in OR ControllerFailed
- FC:{ValHigh, ValLow}-out = SAME(FC)-in

**Bus**
- FC EXCEPT {Val_Detectable, Omission}-OP = SAME(FC)-in
- Omission-OP = Omission-in OR BusFailed
- Val_Detectable-OP = EMI

**TTP/C Receiver**
- Omission-out = ANY(FC) EXCEPT {ValHigh, ValLow}-in OR ControllerFailed
- FC:{ValHigh, ValLow}-out = SAME(FC)-in

**Wheel brake**
- Omission-out = Omission-ALL(IP) OR BrakeFailed
- FC EXCEPT {Omission}-out = SAME(FC)-ANY(IP)

To explain, the pedal has three possible output deviations: an omission of output, caused by an internal failure of the pedal, or a value deviation, either high or low, caused by a corresponding bias in the pedal. Commission failures (unexpected output) are treated as value deviations where the expected value was 0 for the purposes of this case study.

The TTP/C sender reacts to these failures in different ways: value failures are simply propagated, while omissions are caused not only by other omissions but also by any other failure class except value failures. This reflects the ‘fail-silent’ pattern of behaviour in the TTP/C design – any detectable failure results in an omission of output. The TTP/C sender will also omit output if it suffers an internal failure.

The buses follow the simple ‘propagator’ pattern for most failure classes; the exceptions are omissions, which can also be caused by internal bus failure, and detectable value deviations caused by electromagnetic interference (EMI). However, undetectable value failures are simply propagated.

The wheel brakes themselves can suffer from an omission of braking caused by an omission of signal at all inputs (i.e., from both TTP/C receivers) or an internal brake
failure and also from a value failure (i.e., excess or insufficient braking) caused by a value failure at any input.

Lastly, a final, ‘virtual’ component – ‘CarBraking’ – was added to represent the braking function of the car, and this was connected to all four wheel brakes like so:

<table>
<thead>
<tr>
<th>CarBraking</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>NoBrake4-out</td>
<td>= Omission-ALL(IP)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NoBrake3-out</td>
<td>= Omission-MAJ(IP)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NoBrakeRear-out</td>
<td>= Omission-RearLeft AND Omission-RearRight</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NoBrakeFront-out</td>
<td>= Omission-FrontLeft AND Omission-FrontRight</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NoBrakeDiag-out</td>
<td>= (Omission-FrontRight AND Omission-RearLeft) OR (Omission-FrontLeft AND Omission-RearRight)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NoBrake1-out</td>
<td>= Omission-ANY(IP)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ValHigh4-out</td>
<td>= ValHigh-ALL(IP)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ValLow4-out</td>
<td>= ValLow-ALL(IP)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

These output failures reflect different levels of severity in possible braking failures. These can be seen in Table 1, where the different severity levels are the standard severity classes of IEC-1508 (catastrophic, critical, marginal and insignificant).

**Table 1** Effects of failures of the braking system

<table>
<thead>
<tr>
<th>Failure</th>
<th>Effect</th>
<th>Severity</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>NoBrake4</td>
<td>Loss of braking in all wheels</td>
<td>Catastrophic</td>
<td>100% loss of braking function</td>
</tr>
<tr>
<td>NoBrake3</td>
<td>Loss of braking in three wheels</td>
<td>Catastrophic</td>
<td>80% loss of braking function</td>
</tr>
<tr>
<td>NoBrakeFront</td>
<td>Loss of braking in front wheels</td>
<td>Catastrophic</td>
<td>65% loss of braking function, 30% loss of car stability</td>
</tr>
<tr>
<td>NoBrakeRear</td>
<td>Loss of braking in rear wheels</td>
<td>Critical</td>
<td>35% loss of braking function, 30% loss of car stability</td>
</tr>
<tr>
<td>NoBrakeDiag</td>
<td>Loss of braking in diagonally opposite wheels</td>
<td>Critical</td>
<td>50% loss of braking, 15% loss of stability, 15% loss of steering</td>
</tr>
<tr>
<td>NoBrake1</td>
<td>Loss of braking in single wheel</td>
<td>Critical</td>
<td>Marginal unless car is braking while on a curved trajectory, in which case it may drift off course</td>
</tr>
<tr>
<td>ValLow4</td>
<td>Insufficient braking in all wheels</td>
<td>Catastrophic</td>
<td>Major loss of braking function</td>
</tr>
<tr>
<td>ValHigh4</td>
<td>Excess braking in all wheels</td>
<td>Critical</td>
<td>Excess braking can lead to loss of steerability if wheels lock, but stability is mostly maintained</td>
</tr>
</tbody>
</table>

Source: Papadopoulos and McDermid (1999)

Note that there are only two value failures included, those affecting all four wheels, because in this model it is not possible for a single wheel to experience a value failure as there is a single common cause for each type of value failure (biased pedal).
4.2 Results

Once fully annotated, the system model was analysed by a prototype version of HiP-HOPS that has been augmented with GFL capabilities. The results obtained are shown below in a simplified FMEA table. Note that this only shows single points of failure: although HiP-HOPS FMEAs also show the effects of combinations of failures, there are too many of these to show here and so they have been omitted.

Table 2  Simplified FMEA showing only single points of failure in the model

<table>
<thead>
<tr>
<th>Component</th>
<th>Failure</th>
<th>Effect</th>
<th>Severity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pedal</td>
<td>BiasedHigh</td>
<td>ValHigh</td>
<td>Critical</td>
</tr>
<tr>
<td></td>
<td>BiasedLow</td>
<td>ValLow</td>
<td>Catastrophic</td>
</tr>
<tr>
<td></td>
<td>PedalFailed</td>
<td>NoBrake1</td>
<td>Critical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NoBrake3</td>
<td>Catastrophic</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NoBrake4</td>
<td>Catastrophic</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NoBrakeFront</td>
<td>Catastrophic</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NoBrakeRear</td>
<td>Critical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NoBrakeDiag</td>
<td>Critical</td>
</tr>
<tr>
<td>TTP sender</td>
<td>ControllerFailed</td>
<td>NoBrake1</td>
<td>Critical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NoBrake3</td>
<td>Catastrophic</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NoBrake4</td>
<td>Catastrophic</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NoBrakeFront</td>
<td>Catastrophic</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NoBrakeRear</td>
<td>Critical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NoBrakeDiag</td>
<td>Critical</td>
</tr>
<tr>
<td>Wheel_FrontLeft</td>
<td>BrakeFailed</td>
<td>NoBrake1</td>
<td>Critical</td>
</tr>
<tr>
<td>Wheel_FrontRight</td>
<td>BrakeFailed</td>
<td>NoBrake1</td>
<td>Critical</td>
</tr>
<tr>
<td>Wheel_RearLeft</td>
<td>BrakeFailed</td>
<td>NoBrake1</td>
<td>Critical</td>
</tr>
<tr>
<td>Wheel_RearRight</td>
<td>BrakeFailed</td>
<td>NoBrake1</td>
<td>Critical</td>
</tr>
</tbody>
</table>

As can be seen from these results, the duplication of the buses and TTP/C receivers has ensured that none of these components is a single point of failure. The full results show that it requires a failure of either both buses, both receivers at the same wheel, or one receiver and one bus to cause a brake failure at any given wheel.

However, the results also show two other important facts: that there is no protection from value/commission failures and that the TTP/C sender is a single point of failure. The latter can be solved by replicating the sender using the redundancy pattern, so that the system can continue to function even if one sender fails. Regarding value and commission failures, although there is no protection from these, a commission or reduction of braking in some wheels – but not all – is actually more severe than a failure in all four wheels, as it can lead to a greater loss of stability (e.g., if both left wheels brake excessively, the car could veer off in that direction, whereas if all four wheels are affected equally, the driver will maintain more control). An additional strategy is to make use of the ‘voter’ pattern, increasing replication in the system and allowing the wheel brakes to filter out partial value errors by comparing inputs and discarding those that do not agree with the majority.
4.3 Benefits of reuse

Using standard HiP-HOPS expressions would result in a lot of annotations for what is a very small system – but this also makes it an excellent candidate for reuse. There are many instances in this system where failure patterns are exhibited – in particular, the ‘propagator’ pattern in the buses, the ‘fail-silent’ pattern of the TTP/C components and the ‘redundancy’ pattern of the dual TTP/C controllers at the wheel brakes. The annotations for all of these components can therefore make use of pre-existing libraries of failure pattern templates.

Furthermore, reuse is possible with components and not just patterns; for example, the annotations for both buses can be reused not only in this system but could probably make use of a generic bus template that can be used in multiple systems. The nine TTP/C controllers all have the same annotations too, and in this case it is possible to make use of inheritance to better leverage the reusability benefits of the GFL. We can create a generic TTP/C controller template to represent the fail-silent behaviour, as described above, by inheriting the fail-silent pattern:

<table>
<thead>
<tr>
<th>PATTERN TEMPLATE:</th>
<th>TTP/C controller</th>
</tr>
</thead>
<tbody>
<tr>
<td>INHERITS:</td>
<td>Fail-silent</td>
</tr>
<tr>
<td></td>
<td>(Fail-Silent: Omission-OP = ANY(FC)-ANY(IP))</td>
</tr>
<tr>
<td>FAILURE LOGIC:</td>
<td>ValHigh-out = ValHigh-in</td>
</tr>
<tr>
<td></td>
<td>ValLow-out = ValLow-in</td>
</tr>
</tbody>
</table>

In this case, the ANY(FC) in the inherited omission output is implicitly replaced with ANY(FC) EXCEPT {ValHigh, ValLow} in light of the overriding value failures. Then the TTP/C sender and TTP/C receivers can all inherit from this as in both cases the TTP/C components will still inherit the value propagation from the generic TTP/C controller and the fail-silent omission behaviour from the fail-silent template.

Using GFL has meant that considerably fewer expressions were required to annotate this model, resulting in a significant improvement in efficiency. GFL can potentially offer substantial benefits in reusing failure data when annotating system models, particularly for large models or systems that make extensive use of common patterns of failure behaviour, such as this one.

5 Conclusions

Reuse of well-established failure information is a valuable technique during the analysis of safety-critical systems. It can bring a number of benefits when properly applied, including reductions in time and effort as well as allowing analysts to make use of expert information even if they themselves are less familiar with the system in question. Reuse is particularly valuable when describing recurring ‘patterns’ of failure behaviour, such as a component that simply propagates failures or a component designed to fail-silent in response to failures. However, there is no formalised method for specifying failure data in a robust and machine-readable manner and this limits what is currently possible when reusing data, particularly when combined with semi-automated analysis techniques such
as HiP-HOPS that rely on logical expressions to describe the local failure behaviour of components.

In this paper, we introduced a possible solution to this problem: the GFL. It provides new capabilities to describe the failure behaviour of modelled components using a mixture of Boolean logic and generalised references to component ports and failure classes. This abstraction makes it possible to write more powerful expressions that can apply equally and accurately to multiple components with the same failure behaviour, even if their interfaces differ or other failure classes or parameters apply. The resulting failure data is then analysable in automatic tools such as the HiP-HOPS tool and to demonstrate this we applied it to a case study of a car braking system annotated using the GFL.

Furthermore, the GFL also incorporates the concept of ‘inheritance’ or specialisation of failure data, providing an additional avenue for reuse: by taking an existing description of failure behaviour and then adding to it, making it more specific to a particular application without having to rewrite all of the failure data. The true benefit of this approach lies in being able to make use of a set of well-defined failure pattern templates and deriving more specialised component failure data from them.

The benefits of GFL were demonstrated in an example brake-by-wire system. From a small set of reusable failure pattern templates, HiP-HOPS was able to automatically derive system fault trees and FMEAs which have highlighted weaknesses in the design of the system and stimulated useful design iteration. GFL offers significant advantages in efficiency and reusability over standard logical expressions when describing component failure behaviour. Its use in conjunction with templates and automated analysis tools like HiP-HOPS can help to rationalise and simplify safety assessment and enable iterations of safety analyses as part of a system design process, incorporating the results of each analysis into the next iteration of the design and ultimately producing a safer and more reliable system.

Acknowledgements

This work was supported by the EU Projects SAFEDOR (Grant 516278) and ATESST-2 (Grant 224442).

References


Capture and reuse of composable failure patterns


