Generalizable safety annotations for specification of failure patterns

Ian Wolforth1, Martin Walker1, Lars Grunske2,∗,† and Yiannis Papadopoulos1,‡,§

1Department of Computer Science, University of Hull, U.K.
2Faculty of ICT, CS3, Swinburne University of Technology, Australia

SUMMARY

Components in programmable systems often exhibit patterns of failure that are independent of function or system context. In this paper, we show that it is possible to capture, and reuse where appropriate, such patterns for the purposes of system safety analysis. We describe a language that enables abstract specification of failure behaviour and define the syntax and semantics of this language. The language extends concepts originally defined in HiP-HOPS, a technique that enables a largely automated form of compositional system safety analysis. The paper describes how this language can be used to describe component failure patterns and demonstrates how it can be applied using a simple fuel system example. The approach is evaluated on a set of retrospective industrial case studies, where data-mining and reverse engineering techniques are applied in order to identify hidden patterns in legacy safety analyses. Results show clear potential for practical use of patterns in HiP-HOPS. We argue that careful specification and reuse of failure patterns in conjunction with a tool that automates Fault Tree and Failure Modes and Effects Analysis can help to simplify complex safety assessments. Copyright © 2010 John Wiley & Sons, Ltd.

Received 21 May 2009; Revised 6 January 2010; Accepted 14 January 2010

KEY WORDS: failure patterns; compositional safety evaluation; dependability; generalizable safety annotations

1. INTRODUCTION

In the current industrial practice [1], the safety analysis of critical engineering systems is performed using a range of classical techniques, which include Fault Tree Analysis (FTA) [2, 3] and Failure Modes and Effects Analysis (FMEA) [4]. During the assessment process, a team of analysts manually applies a combination of these techniques to predict possible hazards, develop risk evaluation models, and devise strategies to mitigate unacceptable levels of risk where necessary [5]. Although there are many commercial tools (e.g. [6]) that automate the quantitative evaluation of predictive failure models like fault trees, the construction of such models and the overall application of safety analysis techniques remain as manual processes performed by expert analysts. For relatively simple systems, this is a manageable process, but with increasing system complexity,
manual analysis becomes expensive and error prone and questions are therefore asked as to its applicability to complex technologies.

New computer-based systems can deliver more advanced safety functions. Numerous studies, for instance, confirm that Electronic Stability Control in vehicles is highly effective in maintaining control and saves lives by significantly reducing the number and severity of crashes [7]. Although such technologies are often introduced precisely in order to improve safety, they also introduce new hazards caused, for example, by classes of failure modes that did not arise in older electromechanical designs. Such failure modes include, for instance, commission failures, i.e. conditions in which functions are being provided in the wrong context of operation (e.g. inadvertent and incorrect application of brakes in a vehicle control system). Trends in the architecture of programmable systems, including increasing density of functions per electronic unit and distribution of functions on networks of embedded components, also raise serious safety concerns. These include the possibility of common cause failures or that of unpredicted dependent failures of critical functions that could be caused by malfunction of non-critical functions. Finally, difficulties in safety assessment are amplified by the increasing scale and complexity of systems which render detailed, consistent, correct and complete safety analysis increasingly difficult to achieve using classical safety analysis techniques. In practice, these difficulties mean that safety analysis is often carried out only once in the course of the design life-cycle, for certification purposes, while it is generally accepted that iterative application from early stages of design contributes to more effective and cost-effective identification of problems and ultimately contributes to safer outcomes.

Key in addressing these problems, we believe, is the establishment of new and improved safety assessment processes in which a degree of automation and reuse of safety analyses become possible [8, 9]. A body of work has developed over the last 15 years in this direction, towards formal and semi-formal languages and techniques that enable specification, composition and application-level reuse of component safety analyses in the context of system wide safety assessment [10–22] (see Section 7 on related work). These techniques are not directly applicable on source code, but they can be usefully applied for analysis of software & system (i.e. combined hardware & software) architectures in the context of a model-based design process. The compositionality and reuse of component safety analyses achieved in many of these techniques are expected to bring in safety assessment similar benefits to those introduced by reuse of well-tested and trusted software components in general software engineering. It should be noted that although concepts that allow reuse of safety analyses are highly desirable, it is also important that such concepts are combined with mechanisms that enable adaptation of reusable analyses to achieve detailed and specific analysis of failures in different application contexts. System and application context are clearly increasingly important as programmable systems become more complex in behaviour and architecture.

Our contribution to this research is the development of a model-based safety and reliability analysis technique that largely automates and simplifies a substantial part of the assessment, the development of fault trees [20] and FMEAs [23]. This technique is known as Hierarchically Performed Hazard Origin and Propagation Studies (HiP-HOPS) [24]. In this approach, system failure models such as fault trees and FMEAs are automatically constructed from knowledge about the topology of the system and local specifications of failure at component level. These models are then used for prediction of dependability (i.e. safety, reliability & availability) of the system to identify design weaknesses and stimulate design iterations. Although HiP-HOPS generally assumes a compositional model of safety analysis, the technique gives the possibility of a very specific analysis in which component failure models can be adapted in the context of the system and application. The language also provides mechanisms for assessment of environmental and other common cause failures which are application specific and can affect whole subsystems or zones in the topology of the system [25]. The technique has been used for analysis of complex programmable systems [25–27] and it has influenced the error modelling concepts of EAST-ADL2, an emerging architecture description language for the design of automotive systems. HiP-HOPS is currently being harmonized with both EAST-ADL2 [28] and Analysis and Design Language’s (AADL) [29, 30].

In HiP-HOPS, 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. For components that perform simple functions and handle the same inputs across applications (e.g. a flow valve), such specifications are reusable and can be reused in the same application or across applications. However, reuse should be done with care and only where this is deemed appropriate, whereas the present mechanisms for enabling such reuse in the language are quite limited. In this paper, the concept of a local, component-level failure specification is extended to enable description and reuse of a more generalized failure behaviour in the form of patterns. Such patterns could in practice be used to capture common types of fault propagation, fail silence and fault tolerance that components are designed to exhibit irrespective of environment and the number or type of inputs handled. With care, patterns can then be reused in circumstances where components exhibit the same pattern of behaviour in different contexts of operation. More specifically, the contributions of this paper are:

- An extension to the failure modelling language of HiP-HOPS that allows specification, inheritance and improved reuse of generalized patterns of failure behaviour.
- Definition of translational semantics for this language.
- An extension to the automated algorithms of HiP-HOPS to enable synthesis of system safety analyses such as fault trees and FMEAs from specifications in that language.
- Demonstration of the concept in an academic case study.
- Practical validation of the introduced concepts on a set of industrial case studies.

The remainder of the paper is organized as follows: in Section 2, we outline failure modelling and safety analysis in HiP-HOPS and present a simplified example fuel system to highlight the potential usefulness of generalized descriptions of failure behaviour or component failure patterns in HiP-HOPS and safety analysis in general. In Section 3, we define the syntax and semantics of the language extension and explain the use of patterns. In Section 4, we revisit the fuel system to show how application of the syntax offers significant advantages in the specification, abstraction and reuse of component failure behaviour. In Section 5, the approach is evaluated on a set of industrial case studies, where the aim was to determine the potential benefits of using patterns by retrospectively applying the pattern language to legacy safety analyses. By eliciting patterns implicit in the analysis, we show that the proposed linguistic extensions improve the capabilities of the language by allowing specification, and to some extent reuse, of failure annotations in much more compact forms. Finally in Section 6, we consider the related work and then proceed to draw tentative conclusions about the potential usefulness and further direction of this work.

2. COMPOSITIONAL SAFETY ANALYSIS AND MOTIVATION FOR GENERALIZATION

2.1. Compositional safety analysis in HiP-HOPS

In HiP-HOPS, a topological model of the system (hierarchical if required to manage complexity) is annotated with formalized logical descriptions of component failures and then used as a basis for the automatic construction of fault trees and FMEAs for the system. Application of the technique can start once a concept of the system under design has been interpreted into a model which identifies components (i.e. functions or architectural elements) and material, energy or data transactions among components. Suitable models for the application of the technique include abstract functional block diagrams, engineering schematics, piping and instrumentation diagrams, hardware descriptions, data flow diagrams and other models commonly used in engineering and software engineering.

HiP-HOPS can be performed on abstract or more detailed models of the system as these are produced and refined in the course of the design life-cycle. This creates opportunities for reuse and refinement of earlier analyses and the ability to achieve a consistent and continuous assessment in the centre of which lies an evolving model of the system itself. At the early stages of design, the model that provides the basis for the analysis can be a block diagram which shows the
functional composition of the system, input/output transactions among functions and the recursive refinement of functions into networks of lower level sub-functions. Later, when functions are allocated to hardware, the model becomes a representation of the physical architecture of the system which shows components such as sensors, actuators, buses and programmable controllers enclosing software architectures hosted by those controllers. Analysis is possible at each stage of refinement, with greater detail in later stages producing correspondingly greater detail in the results.

A HiP-HOPS study of a system under design has three main phases, the first of which is manual whereas the latter two are fully automated:

- System modelling and failure annotation.
- Fault Tree synthesis from system model.
- FTA & FMEA generation.

The first step in the analysis of a system model in HiP-HOPS is the establishment of the local failure behaviour of each component (i.e. function, hardware or software element) in the model. This local failure behaviour describes how the component itself fails and how it responds to failures—propagated by other components in the vicinity—received at its inputs. In particular, it also specifies the effects that internal or propagated failures have on the component’s outputs. This failure behaviour is represented as a set of logical failure expressions which show how output failures of the component can be caused by internal malfunctions and deviations of the component inputs. A variant of Hazard and Operability Studies (HAZOP) is used to identify plausible output failures and then to determine the local causes of such events as combinations of internal component malfunctions and similar types of input failures. At this stage, analysts are free to define and refine the set of failure classes examined in the course of the analysis, depending on context and application.

Failure classes are abstractions that describe the type of input or output deviation in question. In general, deviations fall into one of several main categories: provision failures, including omissions and commissions; value failures, e.g. too high, too low; and timing failures, e.g. late or early. Components can also transform input deviations of one failure class into output deviations of another and this can be a useful strategy for describing fault mitigation schemes; for example, a component that is designed to fail silent in response to error may convert an input deviation of the value or timing failure classes into an omission at its output. However, it is important to note that failure classes are user-defined and that other classification schemes are equally possible. In some cases, a general classification scheme involving omissions and commissions may be sufficient (and may be more transferrable to similar applications), whereas in other cases, more specific failure classes may be required.

In HiP-HOPS, local failure behaviour is annotated at component-level by means of logical expressions; typically there will be at least one expression per component output in order to fully capture its failure behaviour. These expressions can include Boolean operators such as conjunctions (AND) and disjunctions (OR). As an example of a HiP-HOPS annotation, we can describe an omission of output of a simple component using the following expression:

\[ \text{Omission-}o_1 = \text{Omission-}i_1 \text{ OR InternalFailure} \]

This states that a deviation of class omission of output \(o_1\) can be caused either by an internal failure or an omission of input at input \(i_1\).

Generalizing the failure data in this way—relating the behaviour of a component’s outputs to a combination of input deviations and/or internal failures—gives us a more abstract description of that component’s behaviour. The level of detail can also be increased further by specifying the parameters of the deviations. A connection between components can have more than one attribute; for example, a hydraulic link can be defined by the flow, pressure and temperature of material carried through the link. By using parameters, it is possible to restrict a failure class to refer to a specific attribute, e.g. an unexpectedly high value of the parameter ‘temperature’ at the input might cause a component to overheat.
At early stages of the design, only abstract, logical failure behaviour may be known, and this is used in the analysis phase to determine the possible causes of system failures (this is known as qualitative analysis). At later stages, however, statistical data about failure and repair rates and other parameters of probabilistic distributions may be available, as may estimates of the severity of output deviations and system failures. This information can then be used during the analysis phase to arrive at a figure for the unavailability of each top event, a process known as quantitative analysis, and the unavailability and severity values together can be used as the basis of risk calculations. However, even without this numerical data, a qualitative analysis alone can reveal useful and valuable information about the behaviour of the system, which may inform future design iterations. The results of a qualitative analysis are known as minimal cut sets, which are combinations of events (typically component failure modes) that may cause a system failure to occur. Thus a qualitative analysis will reveal, for example, which failure modes are single points of failure or even common causes for two or more system failures.

Once a component has been annotated with failure data, the component can be stored together with its failure data in a library, so that other components of the same type can use the same failure data or this type of component can be reused in other models with the same failure data. This avoids the designer having to enter the same information many times. As an example of this type of specification, Figure 1 shows the failure data for a generic valve component. The valve has four possible internal failure modes and three possible output deviations at its output (labelled 'b'). The causes of these output deviations include the internal failures but also input deviations at the valve input (labelled 'a'). Failure rates have also been provided for the internal failures.

This specification of internal failure modes is generic in the sense that it does not contain references to the context within which the valve operates. Failure expressions make references only to component malfunctions and input/output ports of the component. The failure behaviour described in these expressions has been derived assuming a simple operation that the component is expected to perform in every application.

Once failure annotations have been inserted for all components in a system model, the topology of the model is used to automatically determine how the local failures specified in those annotations propagate through connections in the model and cause functional failures at the outputs of the system. This global view of failure is captured deductively—moving from effects towards causes—in a set of fault trees which are automatically constructed by traversing the model and by evaluating the local failure expressions encountered during the traversal. The synthesized fault trees are interconnected and form a directed acyclic graph sharing branches and basic events that arise from dependencies in the model, e.g. common inputs which may cause simultaneous dependent failure of hypothetically ‘independent’ functions or physical components. Classical Boolean reduction techniques are applicable on this graph. Thus, in the final phase of a HiP-HOPS analysis, qualitative or quantitative analysis can be automatically performed on the graph to establish whether the

---

Figure 1. Local failure data for a generic valve component.
system meets its safety or reliability requirements. The logic contained in the graph can then be automatically translated into a simple table which is equivalent to a multiple failure mode FMEA [23]. The FMEA records, for each component in the system and for each failure mode of that component, any direct effects on the system and any further effects caused in conjunction with other failure events.

Note that in hierarchical models that record the decomposition of systems, failure annotations may also be inserted at subsystem level to collectively capture the effect of failure conditions that do not necessarily require examination at basic component level. If, for example, a subsystem as a whole is susceptible to some environmental disturbance such as high temperature, flood or electromagnetic interference (EMI), then the effects of this condition can be directly specified with a failure annotation at subsystem level. Such annotations would typically complement other annotations made at the level of enclosed components to describe aspects of failure behaviour at this level (e.g. mechanical and electrical failure modes of each component). In general, when examining the causes of failure at an output of a subsystem, the fault tree synthesis algorithm of HiP-HOPS creates a disjunction between any failure logic specified at sub-system level and logic arising from the enclosed lower levels. This feature makes HiP-HOPS a truly hierarchical approach to the analysis of complex systems. Note that the same concept can be used for the analysis of programmable components, e.g. controllers which enclose software architectures. In such cases, annotations that show the effect of hardware failures on outputs of the controller can be automatically combined with annotations that show the effect of failures propagated through the enclosed software on the same outputs.

In HiP-HOPS, interpretation of the synthesized fault trees, their minimal cut sets and the FMEA helps to identify design weaknesses and initiate design changes. To ensure that design changes do not introduce any new hazardous failure modes, re-establishment of the failure behaviour of the system via iteration of safety analysis should follow such design changes. Clearly, the ability to rapidly iterate this process ultimately defines the ability to effectively manage the evolution of the design in safety assessment. The automated algorithms of HiP-HOPS for the synthesis of fault trees and FMEAs clearly help in this direction. However, the ability to effectively apply the method on evolving designs is dependent upon the ability to annotate each design iteration with appropriate failure data. This makes it important to have failure annotations that are as flexible and reusable as possible; the more generalizable a failure expression is, the more easily it can be reused in later iterations or even different applications without extensive modifications or full re-annotation.

2.2. Case study

As an example of how HiP-HOPS can be applied to a system, and to demonstrate the benefits that generalizable descriptions of failure behaviour can introduce, consider the simplified fuel system in Figure 2. The system is designed to use double paths throughout for redundancy. Fuel is transferred as long as at least one path remains functioning and sufficient power is available. Control is provided by a computer that sends signals to the fuel pumps over replicated signal buses; as long as these signals are present, the pumps will continue to operate. The signals pass through an embedded logic system that can compensate for a permanent failure of one bus by synthesizing the missing control signals from the other bus; thus although an omission may propagate through a single bus, the logic ensures that only omissions in both buses will propagate to the pumps. However, the circuit cannot correct commission errors (unintended transmission of signal), which will propagate to the pumps.

The computer and pumps are powered by a series of redundant power supply units (PSU) routed through a single power bus. The actual nature of the PSUs, or indeed their final number, is not precisely defined in this iteration of the design; the system is meant to be specified in such a way that additional power sources can be added later if necessary. Failure of an individual PSU should be tolerated, but multiple PSU failures may result in total power loss. Note that temporary failures are not as important in this type of system architecture. The rationale behind the design is that if one flow suffers a problem, e.g. an omission of the control signal or a blockage, fuel continues to
Table I. HiP-HOPS failure expressions for the fuel system.

<table>
<thead>
<tr>
<th>Component</th>
<th>Failure Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>PSUs</td>
<td>Omission-out = PSU_failed</td>
</tr>
<tr>
<td>Power Bus</td>
<td>Omission-out1 = Omission-in1 AND Omission-in2 AND Omission-in3</td>
</tr>
<tr>
<td></td>
<td>Omission-out2 = Omission-in1 AND Omission-in2 AND Omission-in3</td>
</tr>
<tr>
<td></td>
<td>Omission-out3 = Omission-in1 AND Omission-in2 AND Omission-in3</td>
</tr>
<tr>
<td>Computer</td>
<td>Omission-out1-sig = Omission-power OR computer_failure</td>
</tr>
<tr>
<td></td>
<td>Omission-out2-sig = Omission-power OR computer_failure</td>
</tr>
<tr>
<td></td>
<td>Commission-out1-sig = memory_stuck</td>
</tr>
<tr>
<td></td>
<td>Commission-out2-sig = memory_stuck</td>
</tr>
<tr>
<td>Data Bus</td>
<td>Omission-out-sig = Omission-in-sig OR bus_failed</td>
</tr>
<tr>
<td></td>
<td>Commission-out-sig = Commission-in-sig</td>
</tr>
<tr>
<td>Logic</td>
<td>Omission-out-sig = Omission-in1-sig AND Omission-in2-sig</td>
</tr>
<tr>
<td></td>
<td>Commission-out-sig = Commission-in1-sig OR Commission-in2-sig</td>
</tr>
<tr>
<td>Valves</td>
<td>Omission-out = Omission-in OR stuck_closed</td>
</tr>
<tr>
<td></td>
<td>Commission-out = Commission-in OR stuck_open</td>
</tr>
<tr>
<td>Pumps</td>
<td>Omission-out = Omission-in OR Omission-control OR Omission-power OR stuck_closed</td>
</tr>
<tr>
<td></td>
<td>Commission-out = Commission-control OR stuck_open</td>
</tr>
</tbody>
</table>

be pumped along the other path. In addition, temporary failures can be seen as less severe versions of permanent failures (e.g. omissions, commissions) and thus the effects of temporary failures are included in an analysis of permanent versions of those failures.

We have focused only on certain classes of failures (namely Omission and Commission) for simplicity, and therefore the only system failure modes we have considered are an omission of fuel flow and an unexpected commission of fuel flow. Because of the dual pumps, the omission can only occur if both pumps cease pumping fuel, whereas the commission can occur if either pump suffers a commission. Because failures can propagate, an omission of flow from a pump can in turn be caused by power loss, an omission of input (either of the input flow or the control signal), or a mechanical failure of the pump itself. This failure behaviour is described in Table I.
Although the HiP-HOPS expressions in Table I are sufficient to describe how the fuel system can fail, there are a number of places where the expressions are cumbersome and not easy to reuse. In particular, the annotations for the power and data buses would require full re-annotation if additional components were added to the design. Taking the power bus as an example, there is a clear pattern of behaviour here: the omission of each output is caused by the omission of all inputs. It would be very useful to be able to use a generalized expression to describe this behaviour, rather than having to enumerate every input and every output—a rather onerous task that is not conducive for the rapid evolution of system designs. Generalized expressions could also be used in this example to state that a computer failure will cause omission of all outputs or that the logic component will fail silent when there is omission of all replicated messages.

Moving beyond this example, it is worth noting that many electromechanical and complex programmable components exhibit patterns of failure behaviour that would be useful to capture. Communication buses, for example, typically propagate input failures to their outputs, while many bus failures would lead to identical effects, e.g. omission of all communicated messages. Similarly, a complex component such as the Time-Triggered Protocol (TTP) communication controller [31] is designed to always fail silent in response to detectable omission, commission and timing failures in received messages, and this behaviour is independent from the number, type or relative scheduling of these messages. The formal description of such patterns and their automated adaptation and application in different design contexts, combined with appropriate tool support, would clearly benefit safety engineering.

3. GENERALIZED FAILURE LOGIC

In this section, we propose a well-defined syntax that extends the logical expressions currently used in HiP-HOPS and makes it possible to capture general patterns of failure behaviour across multiple inputs, outputs, failure classes or parameters and automatically interpret them in the context of compositional safety analysis. This syntax is known as Generalized Failure Logic or GFL.

Definition 1 (Generalized Failure Logic (GFL))
Generalized extension to the HiP-HOPS failure annotation language. Can include both generalized inputs and generalized outputs as well as normal Boolean operators.

3.1. Syntax of the GFL

3.1.1. Definitions. The syntax of the GFL extensions is given below in Table II. The extended annotations require a component identifier, an optional INHERIT directive, and then one or more generalized expressions. In this sense, the expressions can be simple HiP-HOPS expressions that contain no generalized elements, or they can also be more complex constructions involving generalizations over multiple inputs, outputs, parameters and failure classes, mixed with Boolean operators and basic events. Each expression is known as a Generalized Failure Expression (GFE).

Definition 2 (Generalized Failure Expression (GFE))
Generalized form of a HiP-HOPS local failure expression. Each GFE can represent multiple normal expressions.

A GFE consists of two parts: the left-hand part (the output deviation part) and the right-hand part (the cause of the output deviation, usually a combination of input deviations and basic events). Either part may be generalized. Generalized Output Deviations or GODs allow the analyst to generalize over multiple output ports, parameters or failure classes; in doing so, a GFE will represent multiple standard Boolean expressions (one for each port, parameter, or failure class etc.). Similarly, Generalized Input Deviations (GIDs) allow the analyst to generalize over multiple input ports, failure classes or parameters.

A shorthand notation is used to allow the analyst to avoid having to enumerate every port, class or parameter when collective references are made in GODs and GIDs.
Table II. Syntax definition.

Component Definition

(1) ComponentAnnotation ::= ComponentName [InheritDirective] [GFE] +
(2) ComponentName ::= "NAME" ComponentID
(3) InheritDirective ::= "INHERIT" ComponentID
(4) ComponentID ::= Identifier

Generalized Failure Expression DefinitionPattern

(5) GFE ::= GOD "=" InputExpression
(6) GOD ::= OutputFailureClass "=" OutputPort ["=" OutputParam]
(7) InputExpression ::= OrTerm ["OR" OrTerm ]
(8) OrTerm ::= AndTerm ["AND" AndTerm ]
(9) AndTerm ::= ["NOT"] (GID | "(" InputExpression ")" | BasicEvent )
(10) GID ::= InputFailureClass "=" InputPort ["=" InputParam]
(11) BasicEvent ::= Identifier

Failure Class Definition

(12) OutputFailureClass ::= "FC" [Exception] | FCList | FailureClass
(13) InputFailureClass ::= "ANY(" (("FC" [Exception]) | FCList ")") | "SAME(FC)" | FailureClass
(14) FCList ::= "FC:" List
(15) FailureClass ::= Identifier

Port Definition

(16) OutputPort ::= "OP" [Exception] | OutPortList | PortName
(17) InputPort ::= Operator "(" ("IP" [Exception]) | InPortList ")" | PortName
(18) OutPortList ::= "OP:" List
(19) InPortList ::= "IP:" List
(20) PortName ::= Identifier

Parameter Definition

(21) OutputParam ::= "PM" [Exception] | ParamList | ParamName
(22) InputParam ::= Operator "(" ("PM" ParamList ")" | "SAME(PM)" | ParamName
(23) ParamList ::= "PM:" List
(24) ParamName ::= Identifier

Auxiliaries

(25) Exception ::= "EXCEPT" List
(26) List ::= [" Identifier [ "," Identifier ]"]
(27) Operator ::= "ALL" | "ANY" | "MAJ"

Where 'Identifier' is a letter or underscore followed by zero or more letters, numbers, or underscores, which does not contain a case sensitive match of any of the reserved Keywords (NAME, INHERIT, OR, AND, NOT, FC, OP, IP, EXCEPT, ALL, ANY, MAJ) or their long versions. Terminals are presented in "double quotes". For convenience or clarification, some terminals can be replaced as follows: FC = FAILURE CLASS; OP = OUTPUT PORT; IP = INPUT PORT; PM = PARAMETER; NOT= !; OR = +; AND= *

Definition 3 (Generalized Output Deviation (GOD))
The generalized form of an output deviation, appearing on the left-hand side (LHS) of a GFE and representing multiple normal HiP-HOPS expressions.

Definition 4 (Generalized Input Deviation (GID))
The generalized form of an input deviation, consisting of one or more operators and sets of terms. Represents multiple Boolean operators.

In GIDs and GODs, failure class, port and parameter are types which retain the meaning that they have in standard HiP-HOPS as explained in Section 2.2. Depending on the syntax used, a GID or GOD can in practice refer to 'a specific', 'several' or 'every' instance of each type. A generalized type of a GOD or GID is known as a term, and the set of all possible instances of a particular type is called a vector.
Definition 5 (Term)
A generalized element of a GOD or GID. Terms can be ports, parameters or failure classes.

Definition 6 (Vector)
Set of all instances of a given type; OP = all output ports, IP = all input ports, PM = all parameters, FC = all failure classes.

Using appropriate syntax, a term can be a single instance of a type (like ‘out1’ from the power bus), a specific subset or list of instances (e.g. \{out1, out2\}), or the complete set of all possible instances (e.g. all Output Ports: ‘out1’, ‘out2’, and ‘out3’). In the latter case, rather than list all instances, they are referred to using a shortcut: for example, FC is the vector containing all applicable failure classes. Similarly, OP is the vector of all output ports in a component, IP the vector of all inputs and PM the vector of all parameters of a component. Alternatively, we can also use the full vector but specify certain exceptions, e.g. ‘FC EXCEPT {Omission}’ defines a vector containing all failure classes except omission. Several examples of the different collections are shown Table III.

Going back to the power bus example, it is now possible to see how we can describe the behaviour of the power bus with a GFE. We now need only one failure annotation to describe an omission at every output:

Omission-OP = Omission-in1 AND Omission-in2 AND Omission-in3

Note how this failure annotation is inherently more extensible than the original list of HiP-HOPS expressions; it applies equally to a power bus of 10 outputs, for example.

The use of GODs on the LHS (the output deviation side) in this way allows a single failure logic pattern to represent the same information as multiple standard HiP-HOPS expressions; the exact number depends on the interface of the component it is applied to. With the generic vectors replacing specific values in an expression, we can transform an expression into an abstract representation of failure behaviour: a pattern of failure. By abstracting out the context-sensitive information, we can even store and reuse this failure annotation for other components with similar failure behaviour but different contexts (e.g. a different number of outputs). The increased flexibility and possibilities for reuse are the main benefits of using a GFL; however, a GFE does not directly give us the components’ failure behaviour: this is still given by the set of specific failure expressions that the GFE represents, each of which describes how a different output deviation is caused by some combination of input deviations and/or basic events.

To obtain this set of output deviations, we need to instantiate them from the GFE. If we know the contents of the vectors, we can instantiate the full set of expressions from the pattern by substituting each term from a vector in turn. Instantiation is therefore the process by which we derive the standard output deviations represented by a GOD. The resulting set of application-specific failure expressions, the set of it instances of the pattern, can then be used directly in HiP-HOPS for the generation of fault trees and FMEAs.

### Table III. List examples for \(\text{OP}:[\text{out1, out2, out3}]\).

<table>
<thead>
<tr>
<th>EXAMPLE</th>
<th>EQUIVALENT LIST</th>
</tr>
</thead>
<tbody>
<tr>
<td>SINGLE TERM</td>
<td>out1</td>
</tr>
<tr>
<td>LIST OF TERMS</td>
<td>(\text{OP}: [\text{out1, out2}])</td>
</tr>
<tr>
<td>ALL TERMS</td>
<td>(\text{OP})</td>
</tr>
<tr>
<td>SINGLE EXCEPTION</td>
<td>(\text{OP EXCEPT} {\text{out3}})</td>
</tr>
<tr>
<td>LIST EXCEPTION</td>
<td>(\text{OP EXCEPT} {\text{out2, out3}})</td>
</tr>
</tbody>
</table>
on the right-hand side, we need to know which operators apply. Each operator takes a set of terms as its operand and will apply to all the terms in that set.

**Definition 7 (Operator)**

A generalized logical function that applies to a set of terms. Includes ANY, ALL, MAJ and SAME.

For example, in our power bus, an omission of each output is caused by a failure of all inputs, i.e. a conjunction: in1, in2 and in3. We can represent this directly by using the IP (input ports) vector and the ALL operator:

\[
\text{Omission-OP} = \text{Omission-ALL(IP)}
\]

Here, ALL will create a conjunction between the terms in the vector it operates on; in this case, in1 AND in2 AND in3. We need the operator because when a vector is used on the right-hand side, it does not represent multiple expressions; instead, it represents multiple input deviations, i.e. multiple parts of the same expression. An operator therefore allows us to specify how the terms in a vector are combined within an expression. There are two other combining operators available: ANY and MAJ. ANY behaves similarly to ALL, except that it defines a disjunction using OR rather than a conjunction using AND. Note that the conjunction operator can lead to semantic errors if used improperly; for example, using ALL(FC) on the same port could entail a conjunction of mutually exclusive failure classes, e.g. it is not possible to have no value (‘Omission’) and a value out of range (‘Value’) on the same port simultaneously.

The majority operator MAJ is designed to represent voting situations, where a failure only propagates if it occurs at more than half of the input terms. It is in itself a shorthand that represents a disjunction of conjunctions, each containing more than half the total terms of the vector; the minimum number of terms required is known as the **cardinality**, and any MAJ operator is implicitly given the appropriate cardinality when evaluated. For example, if the IP vector had three input ports i1, i2 and i3, then the minimum number of terms that must occur — the cardinality — is 2, and thus MAJ(IP) is evaluated as:

\[
(i1 \text{ AND } i2) \text{ OR } (i1 \text{ AND } i3) \text{ OR } (i2 \text{ AND } i3)
\]

Vectors and operators are inextricably linked in a GID; one is not possible without the other. Operators are not needed in GODs on the LHS because there they represent different output deviations (and thus different expressions), not a combination of input deviations in a single expression.

There is, however, one additional operator which still applies to a vector but does not create a combination: the SAME operator. In certain situations it is useful to define a correspondence between the term of a vector in each instance of an output deviation on the LHS and the current term of that vector in an input deviation on the right-hand side.

As an example, consider a component with simple failure behaviour that merely propagates any failure from its input to its output. We can represent this using multiple expressions as follows:

\[
\text{Omission-out} = \text{Omission-in} \\
\text{Commission-out} = \text{Commission-in} \\
\text{Value-out} = \text{Value-in} \\
\text{etc}
\]

We cannot immediately convert this into a pattern, e.g. FC-out = ANY(FC)-in, because although we do want an instance of the pattern for each failure class, we do not want a combination of all failure classes on the right-hand side—we only want the same failure class. This is the functionality provided by the SAME operator:

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

This means that in each instance of the pattern, SAME(FC) is evaluated as the current failure class of that instance; thus when we instantiate for Omission, SAME(FC) becomes Omission, and so on. The SAME operator can also be applied to parameters (i.e. SAME(PM)) but it cannot be
applied to ports; this is because the output ports of a component may not necessarily correspond to the input ports of a component, either in quantity or in function.

3.1.3. General form of a GFE. The basic form of a GFE is therefore as follows, with the GOD on the left and one or more GIDs on the right:

\[ FC-OP-PM = op1(FC) \- op2(IP) \- op3(PM) \]

where op1 or op3 can be any of the operators in the language (SAME, ANY, ALL or MAJ) and op2 can be ANY, ALL or MAJ. Every vector or set of terms in a GID must have an operator applied to it; however, when using a single term, no operator is needed. Note that a GFE need not contain both a GOD and a GID and may contain only one or the other (in which case the SAME operator is not used). It is also important to remember that this is still part of a normal Boolean expression and therefore there can be multiple GIDs separated by normal Boolean operators, such as AND and OR, together with normal basic events (i.e. internal failures of the component).

For example:

\[ Omission-OP = Omission-IP \lor short\_circuit \]

Using a language of patterns such as this allows a more general approach when describing failure behaviour. It enables us to represent a component’s failure behaviour using a form more akin to the natural language, including keywords that indicate how combinations of input failures contribute to the output deviations and also including the ability to represent multiple output deviations using a single expression. Such GFEs can then be more easily read out in a simpler and more understandable fashion:

_class of failure of parameter at output port is caused by class of failure(s) of parameter(s) at it input port(s) and/or basic event(s)_

Note that this does mean reading the parameters before the ports, e.g. the GFE

\[ FC-out-PM = SAME(FC) \- ALL(IP) \- SAME(PM) \]

would be read as _Each failure class of each parameter at the output is caused by the same failure class of the same parameter at all input ports._

Using two or more operators together results in much more powerful combinations of input deviations; for example, we could specify in a GFE that a given output deviation is caused by _ANY(FC)-MAJ(IP) — any failure class at a majority of input ports_. Defining FC and IP as follows:

FC: \{O, C\}, IP: \{in1, in2, in3\} then the GID is equivalent to:

\[
\begin{align*}
(O-in1 \lor O-in2) & \lor (O-in1 \lor O-in3) \lor (O-in2 \lor O-in3) \\
(C-in1 \lor C-in2) & \lor (C-in1 \lor C-in3) \lor (C-in2 \lor C-in3)
\end{align*}
\]

This type of pattern could be used, for example, in an Adjudicator component that only propagates failures at its output in response to a majority vote of omissions on its input ports:

\[ FC-out = SAME(FC) \- MAJ(IP) \]

It is also possible to use multiple GIDs in the same expression, e.g.:

\[ FC-out = SAME(FC) \- ALL\{in1, in2\} \lor C-ANY\{sig1, sig2\} \]

which can be read as _Each failure class of output ‘out’ is caused either by the same failure class of both of in1 and in2 or by the commission of either sig1 or sig2._

3.2. Inheritance

The final feature of the GFL is _inheritance_, supporting a type of reusability similar to that found in object-oriented software engineering (OOSE). Inheritance in OOSE is a technique that can create new classes from existing classes; the principle is that derived or _specialized_ classes inherit
common characteristics from base classes and can override or augment some characteristics with new, more specialized versions.

Applying this to our failure annotations, the idea is that we can reuse GFEs from a general type of component in a more specific type of component which shares most of its behaviour with the more general component but which, in certain cases, exhibits different, more specific behaviour; for example, perhaps omissions are caused by something different yet all other failure classes behave in the same way. Inheritance therefore gives a description of behaviour which is slightly different from the base component’s failure logic yet still remains ‘compatible’ with the base definition, i.e. it still responds to and generates the same respective input and output deviations (although not necessarily within the same GFE). This is an example of a mechanism similar to polymorphism as found in OOSE, meaning that the more specialized child can also fulfil the role of the parent. In our technique, polymorphism is implicit in all specialized patterns; if a parent class exhibits a certain type of failure at its outputs, or can take a certain type of input deviation, then so does the child (even though the internal details may differ). Not only does this provide a rule for performing the specialization, but the same constraint can also be used for checking correctness (a step which could be automated). For example, as the base component we may have a bus that propagates every failure class from input to output:

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

We can create a specialization of this bus component by inheriting the base behaviour and adding a new description to show that the specialized bus will fail silent in response to input failures of the class ‘Value’:

\[
\text{INHERIT Bus}
\]

\[
\text{Omission-out} = \text{Value-in OR Omission-in}
\]

The specialized component \textit{inherits} the original behaviour of the bus—a direct propagation of failures—but in the case of output deviations of type omission, this original behaviour is \textit{overridden} by new behaviour which states that input deviations of class Value are not propagated (i.e. they are transformed into omissions). In effect, the original GFE will apply for every output deviation except Omission, in which case the new, more specialized pattern applies. This is equivalent to saying:

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

\[
\text{Omission-out} = \text{Value-in OR Omission-in}
\]

3.3. \textit{GFL semantics}

The precise meaning of the language is defined by a transformational semantics specification. This specification defines a set of transformation rules that transform every GFE that can be defined by the syntax (introduced in the previous subsection) into precisely one unique set of simple logical expressions. As the simple logical expressions have a clear semantics, such an unambiguous mapping also gives the new GFE a clear semantics. Before we describe the set of mapping rules for the language, we first have to introduce a formal framework for the failure expressions that are described by a set of component annotations \( C \). Based on the syntax defined in Section 3.1, a component annotation \( c \in C \) can be defined by the following tuple \((C_{ID}, FP_{Rules}, C_{INHERIT})\), where \( C_{ID} \) is a unique component identifier, \( FP_{Rules} \) is defined as a set of failure expression rules, \( C_{INHERIT} \) is a component identifier of the component from which it inherits failure logic expressions. If the component does not inherit any failure logic, then \( C_{INHERIT} = \text{NIL} \). We write \( c.C_{ID}, c.FP_{Rules}, \) and \( c.C_{INHERIT} \) to dereference the elements of the tuple for a component \( c \in C \).

Furthermore, to simplify the transformation rules, there is a set of auxiliary functions that are defined in Table IV. The first three auxiliary functions \( get_{FC}, get_{OP} \) and \( get_{IP} \) return for each component annotation \( c \in C \) the set of identifiers for the failure classes, input ports and output ports. For coherence and compatibility reasons we require that the result of the function \( get_{FC} \)
is identical for all used components, meaning that there is a single set of failure classes for all component annotations $c \in C$. The function $get_{PM}$ takes as input a port identifier and determines the available parameters of this port. The remaining auxiliary functions are basic functions as they can be found in the string packages of common programming languages. Specifically, these functions identify the LHS of a transformation rule by searching for the token $=$, test if a string or a set of strings is contained in a larger string, or convert a simple logical expression or an integer into a string. Furthermore, the function $+$ is used to concatenate two strings.

The transformation rules are presented in Figures 3, 4 and 5. Each rule contains two parts. The first part, which is written above the line, is a set of preconditions for the rule application. If necessary different preconditions are separated by semicolons. For each precondition one or more variables need to be matched with specific instances in the set of components and their failure logic patterns. Only if all preconditions can be matched completely is the transformation rule enabled. As the standard notation for our variables, we use $c, c_1$ and $c_2$ as component variables ($c \in C; c_1 \in C; c_2 \in C$), $s_0, s_1$ and $s_2$ as string variables, $n$ as an integer variable, $fp$ as a variable representing a failure expression rule($fp \in c.FP_Rules)$, $fc$ as a variable representing a failure class identifier, $op$ and $ip$ as a variable representing an output or input port identifier respectively, $pm$ as a variable representing a parameter identifier, $ilist_0$ as string variable representing a comma separated list of identifiers ($[ilist_0]$ returns the cardinality of the list), $logop_0$ and $logop_1$ as variables representing strings in the set \{'NOT', 'AND', 'OR'\} and $logop_0$ as a variable representing strings in the set \{'ALL', 'ANY', 'MAJ', 'MAJ' + Cardinality}. All the identifier variables are per definition strings that do not contain any of the reserved words of the language as defined in Section 3.1.

The second part of each rule, below the line, describes the rewriting that will be done to transform the GFE. Each of the variables matched in the premise of the rules can be used. If a variable is changed, a primed variable $X'$ is used to denote the subsequent value of the variable $X$.

The rules are furthermore grouped into layers that define their execution order. Each rule is identified by two numbers in square brackets separated by a dot, where the first number represents the layer and the second number represents the individual rule number within the layer. Sometimes a rule has alternatives; in this case the alternatives are represented by alphabetic characters as subscripts to the rule number. The first layer contains only one rule ([1.1]). This rule resolves inheritance between the component failure annotations. In the precondition statement, above the line, this rule identifies the inheriting component ($c_1$) and its ancestor ($c_2$); the topmost ancestor

<table>
<thead>
<tr>
<th>Function</th>
<th>Signature</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>getOP</td>
<td>$C \rightarrow String$</td>
<td>Returns the set of output port identifier (as a string, comma separated) of a specific component</td>
</tr>
<tr>
<td>getIP</td>
<td>$C \rightarrow String$</td>
<td>Returns the set of input port identifier (as a string, comma separated) of a specific component</td>
</tr>
<tr>
<td>getFC</td>
<td>$C \rightarrow String$</td>
<td>Returns the set of failure classes identifier (as a string, comma separated) of a specific component</td>
</tr>
<tr>
<td>getPM</td>
<td>$String \rightarrow String$</td>
<td>Returns the set of parameter identifier (as a string, comma separated) for a given input or output port identifier</td>
</tr>
<tr>
<td>getLHS</td>
<td>$String \rightarrow String$</td>
<td>Returns the left-hand side (w.r.t. =) of a failure propagation rule</td>
</tr>
<tr>
<td>toInt</td>
<td>$String \rightarrow Integer$</td>
<td>Converts a string into an integer</td>
</tr>
<tr>
<td>toStr</td>
<td>$LogicExpression \rightarrow String$</td>
<td>Converts a logical expression or an integer variable into a string; $\land$ = 'AND', $\lor$ = 'OR', and $\neg$ = 'NOT'</td>
</tr>
<tr>
<td>$#Str$</td>
<td>$String \times String \rightarrow Boolean$</td>
<td>Checks if a string or an element of a set of strings is contained in another string</td>
</tr>
<tr>
<td>+</td>
<td>$String \times String \rightarrow String$</td>
<td>Translates two strings (describing a comma separated list) into sets and returns the set difference as a string (again comma separated)</td>
</tr>
<tr>
<td>\</td>
<td>$String \times String \rightarrow String$</td>
<td>Translates two strings (describing a comma separated list) into sets and returns the set difference as a string (again comma separated)</td>
</tr>
</tbody>
</table>
must not inherit from any other component. Consequently, cyclic inheritance relations are not
allowed. If all preconditions can be matched, the rule deletes the inheritance relationship
\( c_1 \prec_{\text{INH}} c_2 \).

Furthermore, the rule adds to the inheriting component \( c_1 \) all failure logic expressions of
the ancestor \( c_2 \) that do not have a LHS that is already present in this component.

The rules of the second layer rephrase constructs with the keywords \( \text{FC}, \text{FC EXCEPT}, \text{OP}, \text{OP EXCEPT}, \text{IP}, \text{IP EXCEPT} \) with their List Counterparts \( \text{FC}:[\text{List}], \text{OP}:[\text{List}] \) and \( \text{IP}:[\text{List}] \). For these rules, the relations described in Table III are used. As an example, the rule [2.1] tries
to match a component \( c \) and a failure logic expression \( fp \) of that component that starts with the
keyword \( \text{FC} \) followed by a '-' and an arbitrary String \( s_0 \). If matched, the rule replaces this failure
logic expression by a new one using the list equivalent \( \text{FC} : \{ \text{list} \} \), where the list is defined by
the possible set of failure classes for this component \( \text{get FC}(c) \).

The rules of the second layer rephrase constructs with the keywords \( \text{FC}, \text{FC EXCEPT}, \text{OP}, \text{OP EXCEPT}, \text{IP}, \text{IP EXCEPT} \) with their List Counterparts \( \text{FC}:[\text{List}], \text{OP}:[\text{List}] \) and \( \text{IP}:[\text{List}] \). For these rules, the relations described in Table III are used. As an example, the rule [2.1] tries
to match a component \( c \) and a failure logic expression \( fp \) of that component that starts with the
keyword \( \text{FC} \) followed by a '-' and an arbitrary String \( s_0 \). If matched, the rule replaces this failure
logic expression by a new one using the list equivalent \( \text{FC} : \{ \text{list} \} \), where the list is defined by
the possible set of failure classes for this component \( \text{get FC}(c) \).

These rules cannot be part of the second layer, because to execute these rules, first the related input and output ports have to be clearly identified. For output ports this is only
possible after the rules of the third layer have been executed, and input ports cannot be clearly
identified before the completion of layer six.

Because of the pre-processing in layers two and four, three rules are sufficient to resolve all
the remaining high-level concepts in the output deviations of the failure logic. These rules resolve
Figure 5. Transformation rules for input deviation related constructs and inheritance.

Layer 5: Input Deviations (ANY Operator) for Failure Classes, Input Ports and Parameters

\[ f_p = s_0 + lop_0 + \text{"ANY(FC : \{i + ilist0 + \}"") + s_1 + lop_1 + s_3 \land \text{"NOT", "AND", "OR"} \notin set, s_1 \]

\[ f_p' = s_0 + lop_0 + \text{"(i + ilist0 + \}"") + s_1 \]

\[ f_p = s_0 + lop_0 + s_1 + \text{"ANY(IF : \{i + ilist0 + \}"") + s_2 + lop_1 + s_3 \land \text{"NOT", "AND", "OR"} \notin set, s_2 \]

\[ f_p' = s_0 + lop_0 + \text{"i"} + lop_1 + s_3 \]

\[ f_p = s_0 + lop_0 + s_1 + \text{"ANY(IF : \{i + ilist0 + \}"") + s_2 + lop_1 + s_3 \land \text{"NOT", "AND", "OR"} \notin set, s_2 \]

Layer 6: Input Deviations (MAJ Operator) for Failure Classes, Input Ports and Parameters

\[ f_p = s_0 + \text{"MAJ"} + \text{"(IP : \{i + ilist0 + \}"") + s_1 \]

\[ f_p' = s_0 + \text{"MAJ"} + \text{\(\left[\frac{k_{\text{ma}}}{2} + 1\right]\)} + \text{"(IP : \{i + ilist0 + \}"") + s_1 \]

\[ f_p = s_0 + \text{"MAJ"} + \text{"(PM : \{i + ilist0 + \}"") + s_1 \]

\[ f_p = s_0 + lop_0 + s_1 + \text{"MAJ"} + s_2 + \text{"(PM : \{i + ilist0 + \}"") + s_1 \land ilist0 \neq \emptyset \land \text{"NOT", "AND", "OR"} \notin set, s_1 \]

Layer 7: Input Deviations (ALL Operator) for Failure Classes, Input Ports and Parameters

\[ f_p = s_0 + lop_0 + s_1 + \text{"ALL(IP : \{i + ilist0 + \}"") + s_2 + lop_1 + s_3 \land \text{"NOT", "AND", "OR"} \notin set, s_1 \]

\[ f_p' = s_0 + lop_0 + \text{\(\left[\frac{k_{\text{ma}}}{2} + 1\right]\)} + \text{"(IP : \{i + ilist0 + \}"") + s_1 \]

Layer 8: Input Deviations (SAME Operator)

\[ f_p = fc + s_0 + \text{"SAME(FC)"") + s_1 \]

\[ f_p' = fc + s_0 + fc + s_1 \]

Layer 9: Inheritance

\[ c_1 \in C_1, c_2 \in C_2, c_3 \in inh \Rightarrow c_3 = c_1 \land c_2 \land \text{NIL} \]

\[ c_1, c_2, c_3 \in inh \land \text{NIL} \Rightarrow c_1 \text{FF}\text{Init} := c_1 \text{FF}\text{Init} \}

\{mhr \in c_2 \text{FF}\text{Init}, \forall old \in c_1 \text{FF}\text{Init}, \text{getLHS}(mhr) \neq \text{getLHS}(old) \}

Figure 5. Transformation rules for input deviation related constructs and inheritance.
null
Table V. Possible advanced keywords and their reduction rules.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Reducing Rule (Output Dev.)</th>
<th>Reducing Rule (Input Dev.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PM</td>
<td>[3.1]</td>
<td>Not Allowed</td>
</tr>
<tr>
<td>PM:[List]</td>
<td>[4.1]</td>
<td>See Table VI</td>
</tr>
<tr>
<td>PM EXCEPT[List]</td>
<td>[3.2]</td>
<td>Not Allowed</td>
</tr>
<tr>
<td>IP</td>
<td>Not Allowed</td>
<td>See Table VI</td>
</tr>
<tr>
<td>IP:[List]</td>
<td>Not Allowed</td>
<td>See Table VI</td>
</tr>
<tr>
<td>IP EXCEPT[List]</td>
<td>Not Allowed</td>
<td>See Table VI</td>
</tr>
<tr>
<td>OP</td>
<td>[1.3]</td>
<td>Not Allowed</td>
</tr>
<tr>
<td>OP:[List]</td>
<td>[2.2]</td>
<td>Not Allowed</td>
</tr>
<tr>
<td>OP EXCEPT[List]</td>
<td>[1.4]</td>
<td>Not Allowed</td>
</tr>
<tr>
<td>FC</td>
<td>[1.1]</td>
<td>See Table VI</td>
</tr>
<tr>
<td>FC:[List]</td>
<td>[2.1]</td>
<td>See Table VI</td>
</tr>
<tr>
<td>FC EXCEPT[List]</td>
<td>[1.2]</td>
<td>See Table VI</td>
</tr>
<tr>
<td>SAME(FC)</td>
<td>Not Allowed</td>
<td>[8.1]</td>
</tr>
<tr>
<td>SAME(PM)</td>
<td>Not Allowed</td>
<td>[8.2]</td>
</tr>
<tr>
<td>ALL</td>
<td>Not Allowed</td>
<td>See Table VI</td>
</tr>
<tr>
<td>ANY</td>
<td>Not Allowed</td>
<td>See Table VI</td>
</tr>
<tr>
<td>MAJ</td>
<td>Not Allowed</td>
<td>See Table VI</td>
</tr>
<tr>
<td>MAJ n</td>
<td>Not Allowed</td>
<td>See Table VI</td>
</tr>
</tbody>
</table>

of the list. Consequently, these rules can also only be applied a finite number of times. As all the rules in each layer can only be applied a finite number of times, the rewriting system terminates.

**Proposition 2**
The rewriting system is complete, meaning that each advanced language concept will be eliminated and each input can be reduced to basic Boolean expressions.

To show completeness we have to show that each advanced language keyword that is defined in the syntax in Section 3.1 is removed from the failure logic expressions. All possible keywords that are not allowed in basic Boolean expressions are enumerated in Tables V and VI and rules are provided to eliminate these keywords. Table V indicates the rules that eliminate each allowed advanced language keyword in the output deviation of a failure logic expression. For input deviations, only the reduction rules for the keywords SAME(FC) and SAME(PM) are given. The remaining failure class, input port and parameter related keywords can occur in any combination with the keywords ALL, ANY, MAJ and MAJ n. The reduction rules for each combination are given in a matrix in Table VI. The remaining keyword Inheritance is resolved in rule [1.1].

As a result, the transformation system is able to eliminate each possible advanced keyword and replace them with either simple Boolean expressions or constructs that will be replaced with simple Boolean expressions in the following layers. However, to finally prove completeness, each of the rules must be also enabled in their respective layers. To show this, we have analysed the rules for critical pairs and dependencies. The results of this analysis are given in Table VII. An analysis of these dependencies indicates that no rule is dependent on a rule application in a future layer. Additionally, only the rule [1.1] which is dependent on itself and the rules in layer seven have dependencies with rules in the same layer. However, these rules force the sequential application of the subsequent rules and eventually the related advanced keywords get resolved. As an example, the rule [1.1] is continuously applied from the topmost ancestor component annotation until all inheritance relations are finally resolved.

Based on this argumentation, the proposition is true and the rewriting system is complete. Furthermore, the final outcome is a set of component annotations which contains only basic Boolean expressions.

**Proposition 3**
For each input to the rewriting system there is an unambiguous output.
Table VI. Possible combinations of advanced keywords in input deviations and their reduction rules.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>ALL</th>
<th>ANY</th>
<th>MAJ</th>
<th>MAJ n</th>
</tr>
</thead>
<tbody>
<tr>
<td>PM:{List}</td>
<td>[7.3_{a-d}]</td>
<td>[5.3_{a-d}]</td>
<td>[6.3_{a-d}]</td>
<td>[6.9_{a-d}]</td>
</tr>
<tr>
<td>IP</td>
<td>[1.7]</td>
<td>[1.7]</td>
<td>[1.7]</td>
<td>[1.7]</td>
</tr>
<tr>
<td>IP:{List}</td>
<td>[7.2_{a-d}]</td>
<td>[5.2_{a-d}]</td>
<td>[6.2_{a-d}]</td>
<td>[6.7_{a-d}]</td>
</tr>
<tr>
<td>IP EXCEPT{List}</td>
<td>[1.8]</td>
<td>[1.8]</td>
<td>[1.8]</td>
<td>[1.8]</td>
</tr>
<tr>
<td>FC</td>
<td>[1.5]</td>
<td>[1.5]</td>
<td>[1.5]</td>
<td>[1.5]</td>
</tr>
<tr>
<td>FC:{List}</td>
<td>[7.1_{a-d}]</td>
<td>[5.1_{a-d}]</td>
<td>[6.1_{a-d}]</td>
<td>[6.5_{a-d}]</td>
</tr>
<tr>
<td>FC EXCEPT{List}</td>
<td>[1.6]</td>
<td>[1.6]</td>
<td>[1.6]</td>
<td>[1.6]</td>
</tr>
</tbody>
</table>

Based on the previous propositions we know that the rewriting system terminates and produces a set of component annotations which only contains basic Boolean expressions. However, we currently do not know if this output is unambiguous, meaning that the rewriting system always returns a semantically similar output. To prove that this proposition is true, we have to show that the rewriting system is confluent or more specifically that all rewriting rules within one layer are confluent. Two rewriting rules \( \frac{a}{c} \) and \( \frac{b}{c} \) are confluent if, for all possible sets of component annotations \( C_1, C_2 \) and \( C_3 \), where \( C_1, \frac{a}{c} C_2 \) and \( C_1, \frac{b}{c} C_3 \), then there exists a set of component annotations \( C_4 \) such that \( C_2, \frac{b}{c} C_4 \) and \( C_3, \frac{a}{c} C_4 \). In other words the order of rule application does not affect the final outcome. This definition of confluence can also be extended to derivation sequences [32]. To show confluence we reuse the results of the critical pair analysis in Table VII. Based on this analysis, there are only critical pairs of rules in layers one and seven. In both cases the conflicting rules imply a sequential application order. As an example rule [1.1] can be only sequentially applied from the topmost ancestor to the leaf component annotations in the inheritance hierarchy. Owing to this sequential execution of the rules and the absence conflicting rules in the other layers, the results are unambiguous.

Example:
Given a component annotation with the following characteristics \( \text{FC:}[O,C], \text{IP:}[\text{in1, in2, in3}], \text{OP:}[\text{out1, out2}], \text{PM:}[x,y], C_{\text{INHERIT}}=\text{NIL} \) and failure logic expression:

\[
\text{FC:}[O,C]-\text{OP EXCEPT}[\text{out2}]-\text{PM:}[x,y]=\text{ANY (FC:}[O,C])-\text{ALL (IP)}-\text{SAME (PM)}
\]

As there is no inheritance defined for this example, rule [1.1] does not need to be applied. This failure logic expression will be translated into the following failure logic expression after an application of the rules in layer 2 (more specifically rules [2.4 and 2.7]).
Next, the rules [3.1] and [3.2] in layer 3 are executed, resulting in the following two new failure
logic expressions. Please note that since the list for output ports contains only one element, only
one new expression is created.

\[
\begin{align*}
O\text{-}\text{out1}\text{-}\text{PM} &: \left\{x, y\right\} = \text{ANY}(FC:\{O, C\}) - \text{ALL}(IP:\{in1, in2, in3\}) - \text{SAME}(PM) \\
C\text{-}\text{out1}\text{-}\text{PM} &: \left\{x, y\right\} = \text{ANY}(FC:\{O, C\}) - \text{ALL}(IP:\{in1, in2, in3\}) - \text{SAME}(PM)
\end{align*}
\]

The rules in layer four are not enabled for this failure logic expressions, hence we can go directly
to layer five and the result is the following:

\[
\begin{align*}
O\text{-}\text{out1}\text{-}x &= \text{ANY}(FC:\{O, C\}) - \text{ALL}(IP:\{in1, in2, in3\}) - \text{SAME}(PM) \\
O\text{-}\text{out1}\text{-}y &= \text{ANY}(FC:\{O, C\}) - \text{ALL}(IP:\{in1, in2, in3\}) - \text{SAME}(PM) \\
C\text{-}\text{out1}\text{-}x &= \text{ANY}(FC:\{O, C\}) - \text{ALL}(IP:\{in1, in2, in3\}) - \text{SAME}(PM) \\
C\text{-}\text{out1}\text{-}y &= \text{ANY}(FC:\{O, C\}) - \text{ALL}(IP:\{in1, in2, in3\}) - \text{SAME}(PM)
\end{align*}
\]

The following steps are identical for all four formulae, and hence for simplicity the steps are only
explained for the first failure expression \((O\text{-}\text{out1}\text{-}x)\). In layer six only the rule \([6.1_a]\) (which is
an alternative to the rule \([6.1_a]\)) can be applied, resulting in the following failure logic expression:

\[
O\text{-}\text{out1}\text{-}x = O\text{-}\text{ALL}(IP:in1, in2) - \text{SAME}(PM) \text{ OR } C\text{-}\text{ALL}(IP:in1, in2, in3) - \text{SAME}(PM)
\]

In layer seven no rule can be applied and the application of the rules \([8.2_b]\) and \([8.2_c]\) leads to the
following failure logic expression:

\[
O\text{-}\text{out1}\text{-}x = O\text{-}\text{in}1\text{-}\text{SAME}(PM) \text{ AND } O\text{-}\text{in}2\text{-}\text{SAME}(PM) \text{ AND } O\text{-}\text{in}3\text{-}\text{SAME}(PM) \text{ OR } C\text{-}\text{in}1\text{-}\text{SAME}(PM) \text{ AND } C\text{-}\text{in}2\text{-}\text{SAME}(PM) \text{ AND } C\text{-}\text{in}3\text{-}\text{SAME}(PM)
\]

Finally, after several applications of the rule \([9.2]\) we have the final simple logical expression for
the first GFE. The logical expressions for the others \(O\text{-}\text{out1}\text{-}y, C\text{-}\text{out1}\text{-}x, \text{and } C\text{-}\text{out1}\text{-}y\) can be derived in a similar way.

\[
O\text{-}\text{out1}\text{-}x = O\text{-}\text{in}1\text{-}x \text{ AND } O\text{-}\text{in}2\text{-}x \text{ AND } O\text{-}\text{in}3\text{-}x \text{ OR } C\text{-}\text{in}1\text{-}x \text{ AND } C\text{-}\text{in}2\text{-}x \text{ AND } C\text{-}\text{in}3\text{-}x
\]

4. GFL IN PRACTICE

4.1. Pattern templates

Although GFL allows us to produce HiP-HOPS compatible expressions with reduced context
dependence, facilitating their reuse in other applications, there must still be a structured method-
ology for reusing them if \textit{ad hoc} copy and pasting is to be avoided. In the case of GFL, reuse is
governed by \textit{pattern templates}, which contain a description of the GFL expressions and guidelines
on how they can be reused elsewhere. Pattern templates, therefore, have the same type of goals
as templates for design patterns in software engineering, and it is essential to have some kind of
template system for the documentation of safety patterns in order to promote well-managed reuse.
These pattern templates need a consistent structure to ensure that they contain all relevant and
necessary information for the safe reuse of failure behaviour.

As an example of a pattern template, Table VIII describes the template for a simple propagator,
e.g. some kind of simple bus.

A name for the pattern is necessary to support the inheritance functionality in the GFL, allowing
the analyst to reference an existing pattern template rather than simply copying the contents of
the template into a component annotation. To aid in this, the description of the behaviour and
guidance about how to use it helps the analyst ensure that the pattern in question is suitable for the
current purpose, and this is complemented by perhaps one or two examples of instantiated output
Table VIII. Example of a pattern template.

<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>$\text{ANY}(FC) - \text{OP} = \text{SAME}(FC) - \text{ANY}(IP)$</td>
</tr>
<tr>
<td>INSTANTIATED EXAMPLES:</td>
<td>$O\text{-out-signal} = O\text{-in-signal}$, $V\text{-out-signal} = V\text{-in-signal}$, etc.</td>
</tr>
<tr>
<td>RELATED TEMPLATES:</td>
<td>Multiplexer, Demultiplexer, Power Bus</td>
</tr>
</tbody>
</table>

deviations in normal HiP-HOPS format to show what it really represents. Pattern templates can also themselves inherit from other pattern templates, and if so, this is also indicated in the template itself. In addition, all information necessary for the correct instantiation of the GFL patterns should be present in the template, i.e. what failure classes the pattern template was intended to apply to. Finally, the template should also include any relevant references to similar or related patterns, to assist the analyst in choosing the correct pattern template to reuse e.g. if the current one is unsuitable.

Thus with suitable pattern documentation, we can minimize error-prone and inappropriate reuse methods, and more ad hoc approaches such as cut-and-paste can potentially be eliminated from the failure annotations of components.

4.2. Example patterns

We can now present some general cases for reusable patterns, showing by example the definition of general behaviour and the instantiated form they can represent. Each would ideally be represented by a full template, but for conservation of space, they are not included here.

**Direct Propagation of Failure:** $\text{FC-OP-PM} = \text{SAME}(FC) - \text{ANY}(IP) - \text{SAME}(PM)$

In this pattern, any failure is propagated to the outputs; a failure of any input parameter will cause the same failure of the corresponding parameter at the output. This pattern could therefore be used for modelling components that directly propagate input failures to outputs, e.g. multiplexers/demultiplexers, communication buses. Instantiated examples of this pattern are:

- Omission-out-signal = Omission-in-signal
- Value-out-signal = Value-in-signal

**Global Failure in the same mode:** $\text{Omission-OP-PM} = \text{InternalFailure}$

Here, an omission of each output parameter at each port is caused by a single component failure mode. This expression can be used for modelling components in which a single failure mode causes the same effect on all outputs. Omission is a typical effect, but other common failure effects are also possible, e.g. all parameters produced late. Instantiated examples of this pattern are:

- Omission-out-signal = InternalFailure
- Omission-monitor = InternalFailure

**Fail Silent (Transformation of Failure):** $\text{Omission-OP-PM} = \text{ANY}(FC) - \text{ANY}(IP) - \text{SAME}(PM)$

In this pattern, the failure of any input parameter causes an omission of that parameter at all output ports. This expression can be used for modelling components that detect input failures and respond by failing silent, i.e. transform value, timing and commission failures into omissions. The expression implies very strong failure detection but can be modified to reflect more realistic fail silent behaviour. For example, the assumption that every failure class at the input is detectable can be qualified to exclude value failures that are difficult to detect, so that timing and commission
failures are detected and lead to omissions, whereas value failures are undetected and are propagated directly. Instantiated examples of this pattern are:

\[
\text{Omission-out-signal} = \text{Value-out-signal} \\
\text{Omission-control-signal} = \text{Commission-in-signal}
\]

**Standby Recovery:** \( \text{FC-output-PM} = \text{O-primary-SAME(PM)} \) AND \( \text{SAME(FC)} - \text{input-SAME(PM)} \)

Here, any deviation at the single output port is caused by the combination of an omission from the first (primary) input port and the same deviation type at the second input (input). In other words, once the primary component has failed and the standby is active, any failure of the standby’s input is propagated to the output. This GFE can be used for modelling a standby component that monitors a primary component on primary and, when an omission is detected, it takes over to continue the provision of the intended function; after which, any failure at the standby’s input is propagated to its output. Instantiated examples of this pattern are:

\[
\text{Omission-out} = \text{Omission-primary AND Omission-standby} \\
\text{Commission-out} = \text{Omission-primary AND Commission-standby}
\]

**Redundancy on Inputs:** \( \text{FC-OP-PM} = \text{SAME(FC)} - \text{ALL(IP)} - \text{SAME(PM)} \)

In this pattern, any output deviation is caused by a conjunction of the same failure (of the same parameter) at all input ports. For example, low voltage at the output would be caused only by low voltage at all inputs. This expression can be used to model components that can tolerate failures at some inputs, thereby relying on redundant inputs to produce correct output as long as at least one of the redundant inputs is correct. Instantiated examples of this pattern are:

\[
\text{Omission-out-signal} = \text{Omission-in1-signal AND Omission-in2-signal} \\
\text{Commission-out-signal} = \text{Commission-in1-signal AND Commission-in2-signal}
\]

**Voter:** \( \text{Omission-out-PM} = \text{ANY(FC)} - \text{MAJ(IP)} - \text{SAME(PM)} \)

In the voter pattern, an omission of any parameter at the single output port is caused by any deviation of that parameter occurring at the majority of input ports. This expression can be used in the modelling of a fault tolerant electronic component that continues to output a signal as long as the signal is correct at a majority of inputs; otherwise the component fails silent. An instantiated example of this pattern is:

\[
\text{Omission-out} = \text{Value-in1 AND Value-in2 OR Value-in1 AND Value-in3 OR Value-in2 AND Value-in3}
\]

We should note that for this behaviour to be valid we must ensure the independence of inputs, i.e. that the majority of inputs cannot suffer value failures in which they provide the same incorrect input value. This is a key priority when designing a fault tolerant architecture: diverse hardware and software must be used to prevent the possibility of a dependent input value failure propagating at the output of the voter. However, if there is a possibility for such dependent value failure, for correct failure modelling, input value failures should be split into two classes and use of the above pattern should be avoided.

### 4.3. Applying GFL to the case study

We can now demonstrate, within the context of HiP-HOPS, how the GFL can be applied to the fuel system presented earlier and how using GFEs can be beneficial to safety analysis. The GFEs for the fuel system are shown in Table IX and together they represent the propagation of failure through the model.

Using GFEs for these annotations allows much more flexibility; for example, it is possible to define all deviations from the Computer component as two expressions without needing to specify individual ports. Not only does this reduce the workload of the analyst, since they do not have to create expressions describing every deviation of each output port, but the increased level of

---

*Copyright © 2010 John Wiley & Sons, Ltd.*

*Softw. Pract. Exper.* 2010, **40**:453–483

DOI: 10.1002/spe
Table IX. GFL Failure Annotations for the Fuel System.

<table>
<thead>
<tr>
<th>PSUs</th>
<th>Omission-out = PSU_failed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power Bus</td>
<td>Omission-OP = Omission-ALL(IP)</td>
</tr>
<tr>
<td>Computer</td>
<td>Omission-OP-PM = Omission-power OR computer_failure</td>
</tr>
<tr>
<td>Logic</td>
<td>Commission-OP-PM = memory_stuck</td>
</tr>
<tr>
<td>Data Bus</td>
<td>Omission-OP-PM = Omission-ANY(IP)-SAME(PM) OR bus_failed</td>
</tr>
<tr>
<td>Valves</td>
<td>Omission-out = Omission-in OR stuck_closed</td>
</tr>
<tr>
<td>Pumps</td>
<td>Commission-out = Commission-in OR stuck_open</td>
</tr>
</tbody>
</table>

Table X. Results of the analysis.

<table>
<thead>
<tr>
<th>Omission of output flow</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Omission-Fuel.input</td>
<td>No input flow of fuel to the system</td>
</tr>
<tr>
<td>Computer.computer_failure</td>
<td>Internal failure of the computer</td>
</tr>
<tr>
<td>Valve1.stuck_closed AND Valve2.stuck_closed</td>
<td>Both fuel valves stuck in a closed position</td>
</tr>
<tr>
<td>Bus1.failed AND Bus2.failed</td>
<td>Internal failure of both busses</td>
</tr>
<tr>
<td>Pump1.failed AND Pump2.failed</td>
<td>Mechanical failure of both fuel pumps</td>
</tr>
<tr>
<td>Pump1.failed AND Valve2.stuck_closed</td>
<td>Mechanical failure of first pump and second valve stuck closed</td>
</tr>
<tr>
<td>Pump2.failed AND Valve1.stuck_closed</td>
<td>Mechanical failure of second pump and first valve stuck closed</td>
</tr>
<tr>
<td>PSU1.failed AND PSU2.failed AND PSU3.failed</td>
<td>System power loss</td>
</tr>
<tr>
<td>Commission of output flow</td>
<td>Memory error in computer</td>
</tr>
<tr>
<td>Computer.memory_stuck</td>
<td>First valve stuck open</td>
</tr>
<tr>
<td>Valve1.stuck_open</td>
<td>Second valve stuck open</td>
</tr>
<tr>
<td>Valve2.stuck_open</td>
<td>First pump stuck open</td>
</tr>
<tr>
<td>Pump1.stuck_open</td>
<td>Second pump stuck open</td>
</tr>
<tr>
<td>Pump2.stuck_open</td>
<td></td>
</tr>
</tbody>
</table>

abstraction also means that if this component is reused across other systems or designs, the patterns we have constructed would need fewer (or even no) alterations before they could be reused.

Once given the annotated model, HiP-HOPS automatically synthesized fault trees for the two system failures, ‘Omission of output flow’ and ‘Commission of output flow’, and then analysed them to determine the minimal cut sets. The results of this analysis are given in Table X and help us to identify the weak points in the design. In particular, the analysis indicates the single points of failure in the system, such as the event ‘computer failure’—the occurrence of which is sufficient to cause the system to fail, as it propagates through the system to both the pumps. There are also combinations of failures, such as multiple bus or multiple valve failures.

These results could be obtained through the use of normal failure data annotations in HiP-HOPS, but require more time and effort to produce, as shown earlier. In addition, they would not be fully reusable, either in later design iterations or in other applications of the same components. For example, we might look at the results of this analysis, in particular the single point of failure in the computer, and introduce a second computer to remove this flaw. Alternatively, we may choose
to add (or remove) additional power sources. Without GFEs, it would be necessary to re-annotate many of the components, but because GFEs are generic, they are much more flexible, and in most cases need no changes to accommodate extra components.

5. PRACTICAL VALIDATION

To evaluate the benefit of using the pattern-based expressions over standard HiP-HOPS expressions, we developed a simple algorithm to automatically create GFEs from normal local failure annotations in existing HiP-HOPS models and applied it retroactively to a number of system models provided by industrial partners. These models were not modified in any way to facilitate the detection of patterns. The aim was to ascertain the potential improvement that patterns can bring to safety analysis through both the reduction of the number of expressions required to capture failure behaviour and the reduction of the logical complexity of those expressions.

The logical complexity refers to the quantity of logical operators contained within any one expression. The operators we propose in the new syntax can reduce this complexity by compacting large expressions into simpler abstract operations (see Section 3). Quantifying the effect of this type of reduction is not the primary aim of this validation; we believe the greater benefit is in the reduction of the number of expressions, reflecting the improved expressiveness of the GFL and thus the reduced effort required to annotate a system model.

5.1. Algorithm

Reduction can be performed where a set of failure expressions contain common terms. These sets appear, for example, where the behaviour of a component must be defined with many separate expressions. For instance, each deviation the component generates across all its output ports may fall into the same failure classification; the failure class is a common term in the expressions. Hence the set can be substituted for a single abstract representation of the behaviour by replacing the reference to the failure class with a vector term. Other similar reductions are also possible for ports and parameters.

The algorithm used in the conversion process is as follows:

**Step 1: Re-ordering.** The first step is to reorder the logical expressions so that common elements are grouped together. All expressions that refer to the same output port would be grouped together, then within those groups, all expressions with the same output failure class would be grouped, and then the same for parameters.

**Step 2: Reduction.** The second step is to try to reduce the number of expressions. We do this by attempting to replace output ports, failure classes and parameters with the vector or list equivalents. This takes place in two separate phases, the first of which is to detect any potential applications of the SAME operator, by reverse applying the transformation rules [9.1] and [9.2] from Figure 5, thus rewriting for example: $O-o1 = O-i1$ with $O-o1 = \text{SAME}(FC)-i1$. In the second phase, we search for similar failure logic expression and reduce them by reverse applying the rules [3.1], [3.2] and [5.1].

**Step 3: Contraction.** Having reduced the number of expressions for a given component, the next step is to reduce the size of each expression by contracting the input deviation side using operators like ALL and ANY. We can speed up this process with a preliminary reordering of the right-hand side of the expressions so that operators are grouped together, e.g. ‘A AND (B OR C) AND D’ would be reordered to ‘A AND D AND (B OR C)’. The next step of the contraction is to detect any disjunctions/conjunctions that can be contracted and it is repeated until no more contraction is possible. This can be done by reverse applying the semantic rules in layers six and eight. As an example, the following expression $0-i1 \text{ AND } 0-i2 \text{ OR } C-i1 \text{ AND } C-i2$ is first reduced with the rules [8.2] to: $0-\text{ALL}(IP:\{i1, i2\}) \text{ OR } C-i1 \text{ AND } C-i2$ and then by a second application of rule [8.2] to: $0-\text{ALL}(IP:\{i1, i2\}) \text{ OR } C-\text{ALL}(IP:\{i1, i2\})$. There are no more port contractions possible, hence next we look at failure classes and find that we can contract further with rule [6.1] to obtain: $\text{ANY}(FC:\{O, C\})-\text{ALL}(IP:\{i1, i2\})$ which is fully contracted.
Table XI. Results after applying patterns to a number of industrial case studies.

<table>
<thead>
<tr>
<th>Model</th>
<th>No. annotated components</th>
<th>Total expressions</th>
<th>Avg. no. expressions per component</th>
<th>Reduction in expressions via identification of patterns</th>
<th>New total expressions</th>
<th>Avg. no. expressions per component after reduction</th>
<th>Reduction Ratio (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>109</td>
<td>776</td>
<td>7.1</td>
<td>314</td>
<td>462</td>
<td>4.2</td>
<td>41</td>
</tr>
<tr>
<td>B</td>
<td>20</td>
<td>75</td>
<td>3.7</td>
<td>26</td>
<td>49</td>
<td>2.4</td>
<td>35</td>
</tr>
<tr>
<td>C</td>
<td>27</td>
<td>100</td>
<td>3.7</td>
<td>33</td>
<td>67</td>
<td>2.5</td>
<td>33</td>
</tr>
<tr>
<td>D</td>
<td>65</td>
<td>175</td>
<td>2.7</td>
<td>60</td>
<td>115</td>
<td>1.8</td>
<td>34</td>
</tr>
<tr>
<td>E</td>
<td>141</td>
<td>1297</td>
<td>9.2</td>
<td>583</td>
<td>714</td>
<td>5.1</td>
<td>45</td>
</tr>
<tr>
<td>F</td>
<td>85</td>
<td>204</td>
<td>2.4</td>
<td>12</td>
<td>192</td>
<td>2.2</td>
<td>6</td>
</tr>
<tr>
<td>G</td>
<td>1125</td>
<td>5282</td>
<td>4.7</td>
<td>1914</td>
<td>3368</td>
<td>3.1</td>
<td>36</td>
</tr>
</tbody>
</table>

The second phase of STEP 3 is to detect locations where the MAJ operator can be applied. Therefore the rules of layer seven are used.

5.2. Results

By comparing the number of expressions in the original model annotations with the number of expressions after the application of the pattern conversion algorithm, we can gain some insight into how effective the pattern language is. Large reductions in the number of expressions needed will improve the efficiency of the annotation phase (since fewer expressions are needed), provide more possibilities for reuse (since a single GFE can often be used in more than one component, even if the interface changes) and obviously reduce the amount of time needed for the analysis. This can make analyses of complex systems more manageable and can prove very useful during the design process, since GFEs potentially mean that fewer components will need to be re-annotated.

The models used in this study were provided by Germanischer Lloyd, DaimlerChrysler, and Volvo. They include a ship engine cooling (A) system [26], two blow-out prevention systems (B, C) [25] used in offshore platforms, a fueling system (D) [33], an automotive brake-by-wire system (E) [24] and a steer-by-wire system (F) [27]. We also include a collective analysis of unpublished studies, consisting of 20 models of systems (G). The results are shown in Table XI.

As the table shows, most models achieved significant reductions in the number of expressions needed, with a reduction ratio—the percentage of expressions that can be removed by generalizing them—of more than 33%. This means that fewer than two thirds as many expressions were needed when using the GFL.

Only one model, the steering system (F), did not experience significant reduction. This model is annotated for early functional analysis, using small functions with few inputs and outputs and containing only causes of omission and commission failures. As a result, there is a relatively small number of expressions per component (only 2.4) and consequently there is less scope for reducing the number using patterns.

The abstract nature of the GFL provides the greatest benefit when applied to those models that have been fully annotated with expressions that capture the complete range of failure behaviour. In such models, the large degree of duplication between multiple expressions is reduced by using a smaller number of more abstract GFEs instead. Without generalization, the standard HiP-HOPS syntax often leads to a degree of repetition in the expressions and terms and hence this is a good example of a situation where GFEs can be beneficial.

Correspondingly, the greatest level of reduction was seen in the brake-by-wire system model (E), where the total number of expressions was reduced by 45%. In this case the model was thoroughly annotated, with a high number of expressions per component, which leads to greater levels of reduction. The significant reduction achieved is also due to the fact that this model contains many expressions which share a similar logical structure, making it a good candidate for reduction. Finally, it is worth noting that manual inspection of the model revealed that the algorithm had successfully elicited a pattern for a central controller bus, which effectively reduced a large number...
of expressions by stating in GFL that (a) every omission delay or corruption of input messages propagates to the outputs, (b) any bus malfunction causes omission of all messages, and (c) EMI during communication causes detectable corruption of all messages.

We should note that the algorithm presented above may not detect all possible applications of the GFL and may therefore underestimate the extent of the reduction possible. Manual inspections of the analysed models revealed that patterns of fail silence and fault tolerance naturally occurred in the failure annotations given by analysts and the primitives of these patterns in terms of single GFEs were successfully elicited by the algorithm. In the future, the algorithm could be extended to detect patterns that comprise more than one GFE. This can be achieved by searching for known patterns that have been specified by analysts and have been stored in a library of patterns.

The effect of the logical contraction on the complexity of the expressions is comparatively small. For example, in the cooling system (A), of the reduced set of 462 expressions, only 6 benefited from this type of reduction. Any significant simplification in this respect is very much dependent on the type of component—and thus the type of expression—being analysed. Nevertheless, where it can be used, it creates more manageable expressions with a simpler structure.

It is important to note that reverse engineering the models in this way can operate only on the expressions already present; if the GFL were used during the creation of the model, it would be much easier for the analyst to identify applications for reusable patterns (such as a voter, or a standby-redundant system), where the use of operators and vectors is integral to the extensibility and applicability of the GFL. Therefore we would expect the number of expressions needed with GFL compared to standard HiP-HOPS to be reduced even more in practical usage. Furthermore, if the analyst had access to a library of components previously annotated with GFEs, the extent of GFL use would also be greater.

6. RELATED WORK

There are a number of emerging model-based safety analysis techniques aiming at formal specification, failure modelling and reuse [8, 9] of component safety analyses. Broadly speaking, these techniques can be categorized into two classes [34]: semi-formal compositional safety analysis techniques [11–13, 18–20] and adaptations of formal verification techniques that support safety analysis [10, 14–17, 21, 35, 36]. To our knowledge, none of these approaches incorporates linguistic concepts similar to those presented here or capabilities for specification of generalized patterns of failure behaviour. A broader comparison of HiP-HOPS with these approaches, however, will still help to place the work presented in this paper in context.

6.1. Compositional safety analysis techniques

The first compositional safety analysis technique emerged in the mid-90s under the name Failure Propagation and Transformation Notation (FPTN) [11]. FPTN is a graphical notation in which component modules that describe the local generation and propagation of failure can be composed into larger system modules. The concepts introduced by FPTN have influenced subsequent developments including HiP-HOPS [24]. One difference between FPTN and HiP-HOPS is that while failures in FPTN propagate between FPTN modules, in HiP-HOPS failures are simply annotations on the nominal model and propagate via input and output parameters on connections between components. FPTN has remained a notation for describing failure propagation, whereas HiP-HOPS has been enhanced with concepts and algorithms for synthesis of fault trees [20] and FMEAs [23], temporal safety analysis [37] and architectural optimization [27].

Successors of FPTN and HiP-HOPS which define similar concepts of failure propagation include work on Component Fault Trees (CFTs) [18, 38], State-Event Fault Trees (SEFTs) [13, 39], Architecture AADL Error Annex [29] and the Failure Propagation and Transformation Calculus (FPTC) [40].

In CFTs [18], the local failure logic of components is defined in a graph of interconnected modularized fault trees. Although the notation is closer to fault trees, the specification is
6.2. Extensions of formal verification techniques for safety analysis

As an alternative to compositional safety analysis, a number of techniques and tools (including Altarica [10], FSAP-NuSMV [45, 46], Software Deviation Analysis [16, 21], automated FMEA with timedBT [47] and Safety Interfaces [48, 49] and DCCA [50]) have used variants of state-modelling, model-checking and fault simulation as a means of inferring the effects of component failures in a system [21, 51]. Some techniques also allow the analysis of systems with dynamic fault trees, e.g. a recent extension to DCCA [52] and an algebraic method based on DFTs [53].

Formal analysis techniques typically require a conceptually simpler form of component failure modelling than HiP-HOPS in which only failure modes—but not the local propagation of failures from inputs to outputs—must be specified prior to automated analysis (even so, the effort required for the formal modelling of the system in these techniques should not be underestimated). However, this simplification and the consequent higher degree of automation gained by application of these techniques come at additional cost.
First, formal techniques typically define their own language for nominal and failure modelling which is not always fully compatible with other widely used design languages and tools. HiP-HOPS, on the other hand, focuses only on failure modelling and can easily complement design languages focusing on descriptions of nominal behaviour. It has so far been demonstrated to work as an add-on to EAST-ADL2 [28], Matlab Simulink [54], and Simulation X [26] a tool based on Modelica.

Second, the analysis of individual failure modes via simulation or model-checking is computationally expensive and the often forward, inductive nature of the analysis (from causes to effects) creates difficulties, especially when combinations of failures need to be considered. In fault simulation, for example, assuming that there are \(N\) possible component failures in a system, assessment of combinations of \(M\) of those failures requires that the analysis is repeated \(\frac{N!}{((N-M)!*M!)}\) times. For a system that has 1000 failure modes, assessment of the effects of combinations of two failure modes requires that the analysis is repeated approximately half a million times. Although this number can be reduced by carefully exploiting assumptions of independence and by taking advantage of the monotonic properties of failure in coherent failure scenarios, the problem of combinatorial explosion still persists. In fault tree synthesis from compositional mode automata, forward reachability analysis poses combinatorial explosion problems [55], while exhaustive exploration of state space in search of violations of requirements in techniques that use model checking is computationally very expensive [45]. In HiP-HOPS, the analysis of propagation of failures is done by a simple compositional deductive algorithm which links effects on system outputs to causes in the architecture. Synthesis of fault trees is achieved in linear time and is not dependent on the highest order cut set (i.e. the maximum number of failure modes considered in combination that is defined only by the positioning and nesting of AND gates in the error propagation model). Although the subsequent FTA can still be time consuming, overall simplicity has enabled not only application of the technique to large systems but also its combination with computationally greedy heuristics such as Genetic Algorithms for the purpose of architectural optimization with respect to dependability and cost [27].

Finally, it is worth noting that HiP-HOPS has been developed as a dedicated safety analysis technique and as a result it has incorporated primitives that enable advanced probabilistic analyses, including Poisson, Binomial and Weibull calculation models [56], as well as capabilities for common cause and zonal analyses [57]. Although in principle formal techniques could be extended in similar ways, few steps have been taken in this direction.

Overall, we feel that compositional safety analysis and formal verification techniques have different strengths and benefits and can be used in a complementary way in the context of advanced assessment processes. Formal techniques enable verification as well as safety analysis, whereas compositional techniques are scalable and can therefore be used in applications where scalability is an imperative. HiP-HOPS in particular can offer significant benefits for the analysis of safety-critical systems, with features such as the ability to specify failure patterns, generation of multiple failure mode FMEAs and possibilities for architectural optimization.

7. CONCLUSION

In this paper, we discussed a number of difficulties in safety assessment caused by the increasing complexity of modern systems and highlighted the need for safety analysis techniques that can address these difficulties by exploiting common patterns of failure behaviour and enabling potentially reusable component-based specifications of failure. We then proposed linguistic concepts that enable representation and possible reuse of such patterns in the context of HiP-HOPS, an established compositional safety analysis technique.

The issue of a generalizable, reusable representation of failure behaviour is not particular to HiP-HOPS. Several other emerging safety analysis methods use a compositional approach to determine the causes of potential hazards. The results of such analyses on simple components can often be generalized to enable their reuse across designs, where such reuse is possible. In this paper, we
have shown that a greater level of flexibility can be achieved by generating context-free generalized expressions that describe failure behaviour independently of the architecture of the component. Such expressions are better suited to being used as part of an iterative design process, preventing the need to re-annotate components in every annotation. They can also be more easily reused across applications where the same component may receive different types and number of inputs, or more generally where components have the same functionality but different interfaces.

By using the technique described in this paper, it is now possible to capture common patterns of failure behaviour using powerful and compact logical expressions as abstract patterns, which are sufficiently formalized to allow them to be mechanically interpreted for responsible reuse and automatic generation of system safety analyses. By making use of these GFEs, there is the possibility of significant reductions in the time and effort required to annotate a system model in preparation for a HiP-HOPS analysis—in some cases by more than a third. Perhaps just as importantly, these expressions are much better candidates for reuse across different design iterations and even different system models, and hence the true savings would be even greater. Apart from the obvious time benefits, reusing failure logic in this way can also help to avoid errors and omissions in the annotations; by making use of previously stored and validated GFEs, we not only reduce the time needed but also reduce the possibility of errors when creating new expressions for many similar components. Finally, because GFEs are inherently more extensible than standard expressions, use of them makes it easier to use HiP-HOPS as part of an iterative design process, allowing designers to improve and change models to remedy flaws without always having to re-annotate the components in the process. However, although GFL can benefit this process, we should also point out that syntactic structures that enable greater generalization are not in themselves sufficient to enable reuse of failure behaviour. Such reuse will often be limited by the effect of application context and the complexity of component function. Future work on GFL is currently looking into integration of the syntax described here with recent temporal extensions to the syntax of HiP-HOPS that enable specification and analysis of sequential failure scenarios [37].

Overall, the potential benefits from application of this approach on a large scale are substantial and include easing the safety assessment of complex systems, making the examination of effects of design modifications on safety cost-effective and keeping the safety analyses consistent with the design. To realise these benefits, we are currently working towards integration of this work with mainstream architectural description languages including EAST-ADL2 and AADL. The harmonization and integration of emerging model-based design and model-based safety analysis technologies is an important step, we believe, towards addressing the difficulties arising from the increasing complexity of computer-based safety critical systems.

ACKNOWLEDGEMENTS

This work was supported by the EU Projects SAFEDOR (Grant IP-516278) and ATESST-2 (Grant 224442). We thank Germanischer Lloyd (Erich Ruede, Rainer Hamann) and Volvo (Christian Grante) for providing some of the case studies.

REFERENCES


