Component Failure Behaviour: Patterns and Reuse in Automated System Safety Analysis

Yannis Papadopoulos, Audrey Tran, Jean-Marc Faure, Christian Grante

Component Failure Behaviour: Patterns and Reuse in Automated System Safety Analysis

Yiannis Papadopoulos
Department of Computer Science, University of Hull, U.K.

Audrey Tran, Jean Marc Faure
Laboratoire Universitaire de Recherche en Production Automatisée,'Ecole Normale Supérieure de Cachan, France

Christian Grante
Volvo Cars, Sweden

ABSTRACT

Recent work in the area of safety analysis has shown that system Fault Trees and Failure Modes and Effects Analyses (FMEAs) can be automatically derived from a topological model of the system that has been annotated with local, component-level, specifications of failure. In this paper, the concept of a component failure specification is extended to enable description and reuse of generalized *patterns* of failure behaviour that are commonly exhibited by components. A language for the description of such patterns is specified, useful patterns are presented and the use of such patterns is demonstrated on an example of a Time-Triggered system. The paper tentatively concludes that careful reuse of failure patterns in conjunction with automated fault tree and FMEA synthesis algorithms can help to rationalize, and simplify, complex safety assessments.

INTRODUCTION

Empirical evidence shows that the safety of vehicles has improved with the introduction of active safety functions such as antilock braking, traction control and electronic stability programs [1]. It is, therefore, expected that new active safety functions will result to further safety improvements in future vehicle technology. At the same time, there is a growing recognition that new technologies introduce new risks which need to be identified and effectively contained before any new systems are deployed.

Active safety systems may address known safety problems but also introduce new potentially hazardous failure modes. In a traditional design, for example, a commission failure such as the inadvertent application of brakes on a single wheel of the car is impossible. This condition becomes possible, though, in a design that enables independent by-wire control of wheel brakes. Active safety functions that control such brakes are carefully designed to fail silently in case of detected malfunctions. But although the likelihood of commission failures can be reduced via good design, the potential still remains. The severity and probability of occurrence of these and other failure modes likely to arise from the introduction of new technologies in vehicles need to be carefully considered to ensure safe deployment of such technologies.

Although the impact of increasing technological complexity on safety is universal, the issue is perhaps more urgent in the automotive industry which currently undergoes a rapid transition towards distributed embedded vehicle control systems. Such systems bring together a number of currently standalone functions in a common platform where functions are implemented using a pool of shared information and energy resources. In practice, one of the major challenges towards implementation of distributed active safety in vehicles is automatic control of the steering system. Such control shall enable the implementation of a new generation of powerful active safety functions which include side wind compensation, vehicle stability control with steering intervention, lane keeping or changing aids and collision avoidance systems. To enable this type of control, radical changes will need to be made in the steering system. The mechanical link between the steering wheel and the vehicle wheels will almost certainly need to be interrupted and a feedback actuator will be required to provide steering feedback to the driver. Understandably, such radical design changes raise serious safety concerns and demand the thorough safety evaluation of any new design concepts. Potential failure modes must be identified and the effects of these failure modes in the provision of sensitive active safety functions must be established.

Safety concerns in the automotive and other industries are further amplified by the prospect of integration of critical and non-critical vehicle functions on networks of embedded controllers. Indeed, the implication of such integration, i.e. interaction, interoperation and sharing of hardware resources create potential for hazardous common cause failure and unpredicted dependent failure of critical functions that could be caused by malfunction of non-critical functions. Traditionally, such issues were addressed using classical safety analysis such as Fault Tree Analysis (FTA) and FMEA. Today, though, increasing complexity in automotive technology seriously questions the applicability of classical manual risk and safety analysis techniques on new designs. Perhaps the biggest source of difficulties is the manual nature of those techniques, which makes a complete and correct classical safety analysis increasingly more difficult to achieve. Other problems include omissions, errors and inconsistencies in the results from the assessment caused by a selective and fragmented analysis which is often restricted to assumed “critical parts” of the system.

Crucial in addressing these problems, we believe, is the establishment of new and improved safety assessment processes in which composability and reuse of safety analyses becomes possible [2,3]. Such processes are already vaguely prescribed by modern safety standards. The CENELEC railway standards [4], for example, introduce the concept of composable safety cases, according to which the safety case, i.e. the collective evidence of safety of a system, is composed of the safety cases of its sub-systems or components, which in theory could be produced and certified independently. This type of composability in safety analysis is expected to bring similar benefits to those introduced by well-tested and trusted software components in software engineering [5,6].

To achieve this goal, a body of work is already looking into techniques for specification and application-level reuse of component-based safety analyses [7-10]. Our contribution to this work 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 [8,11] and FMEAs [12]. This technique is known as HiP-HOPS (Hierarchically Performed Hazard Origin and Propagation Studies) and enables a largely automated and thus simplified form of compositional safety analysis. 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. This has the beneficial effect of simplifying the analysis whilst keeping the analyses consistent with the design information.

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. In this paper, the concept of a local, component-level failure specification is extended to enable description and reuse of more generalised failure behaviour in the form of patterns. 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. The Time Triggered Protocol (TTP) communication controller [13,14], for instance, is designed to consistently fail silent in response to any timing or commission failures generated by the controller hardware or by the local host. This behaviour is independent of application and justifies the development of a failure pattern for the TTP controller that could be reused across different applications and safety assessments.

In this paper, we highlight the need for more generalized descriptions of failure behaviour or component failure patterns. We also define a language for the description of such failure patterns and explain the use of patterns in conjunction with fault tree and FMEA synthesis algorithms in the context of HiP-HOPS. Finally, we present a number of useful patterns including one that encapsulates the properties of the TTP controller and demonstrate their use in a small example. We discuss the process and results of the analysis and draw tentative conclusions about the potential usefulness and further direction of this work.

COMPOSITIONAL SAFETY ANALYSIS

HiP-HOPS is a model-based semi-automatic safety and reliability analysis technique that uses tabular failure annotations as the basic building block of analysis at component level. In HiP-HOPS, a topological model of the system (hierarchical if required to manage complexity) is first annotated with formalised 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 an engineering model which identifies components 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 of course creates opportunities for re-use of earlier analysis 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 on, 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, busses and programmable controllers enclosing networks of tasks running upon those controllers.

The first step in the analysis of such models in HiP-HOPS is the establishment of the local failure behaviour of each component (i.e. function, hardware or software element) in the model 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) [11] is used to identify plausible output failures such as the *omission*, *commission*, *value* (*hi*, *low*) or *timing* (*early*, *late*) failure of each output and then to determine the local causes of such events as combinations of internal component malfunctions and similar types of input failures. Once this analysis has been completed for all components, and the derived failure expressions have been inserted into the model, the topology of the model is then used to automatically determine how the local failures specified in those expressions propagate through connections in the model and cause functional failures at the outputs of the system. This global view of failure is captured 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 synthesised 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 and recent algorithms for fault tree analysis that employ Binary Decision Diagrams (BDDs) are applicable on this graph. Thus, qualitative analysis (e.g. of abstract functional models) or quantitative analysis (e.g. calculation of system-level failure rates from known failure rates on component-level) can be automatically performed on the graph to establish whether the system meets its safety or reliability requirements. In recent work we have shown that the logic contained in the graph can be automatically translated into a simple table which is equivalent to a multiple failure mode FMEA [12]. The FMEA records, for each component in the system and for each failure mode of that component, any direct effects on the system and 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 like high temperature, flood or electromagnetic interference, 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.

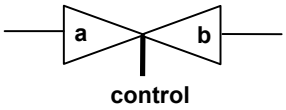
Failure annotations may also optionally include failure and repair rates which, if provided at this stage, are embedded in the synthesised fault trees and can be used to perform probabilistic calculations aimed at prediction of the reliability of the system. Note, though, that such failure rates are not essential and that qualitative application of the technique can still produce useful results. FMEAs, for instance, can indicate components that represent single causes of severe system failures.

IS REUSE POSSIBLE?

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 iterate fast this process ultimately defines the ability to manage effectively 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 also heavily dependent upon the ability to re-use the failure annotations that are required at component level. For this reason, we turn our attention to the question of reuse, and we try to establish whether such reuse is possible and under which conditions.

In addressing the question of reuse, it would be useful to start with the local failure model of a simple component such as the two-way computer controlled valve illustrated in Fig.1. The figure shows the valve as it would typically be illustrated in a plant diagram and records the results of the local safety analysis of the component in two tables that define valve malfunctions and output deviations respectively.

In normal operation, the valve is normally closed and opens only when the computer control signal has a value of a logical one. Valve malfunctions include mechanical failures such as the valve being *stuckOpen* or *stuckClosed*, and blockages caused by debris such as *blocked* and *partiallyBlocked*. For each malfunction, the analysis records an estimated failure rate while the effects of those malfunctions on the output of the valve can be seen in a second table that lists output deviations.



Valve Malfunctions

Failure mode	*Description*	*Failure rate*
blocked	*e.g. by debris*	*1e-6*
partiallyBlocked	*e.g. by debris*	*5e-5*
stuckClosed	*Mechanically stuck*	*1.5e-6*
stuckOpen	*Mechanically stuck*	*1.5e-5*

Deviations of Flow at Valve Output

Output Deviation	*Description*	*Causes*
Omission-b	*Omission of flow*	*blocked or stuckClosed or Omission-a or Low-control*
Commission-b	*Commission of flow*	*stuckOpen or Commission-a or Hi-control*
Low-b	*Low flow*	*partiallyBlocked or Low-a*
Hi-b	*Hi flow*	*Hi-a*
Early-b	*Early flow*	*Early-a or Early-control*
Late-b	*Late flow*	*Late-a or Late-control*

Fig.1. Failure annotations of a computer-operated two-way valve

Here, we can see that an omission of the output flow (*Omission-b*) can be caused by a number of malfunctions such as valve *blocked* or *stuckClosed* and by input failures such as omission of input flow (*Omission-a*) or a value failure in the control signal (*Low-control*). Similarly, commission of the output flow (*Commission-b*) can be caused by a valve malfunction (*stuckOpen*), commission of input flow (*Commission-a*) or a value failure in the control signal (*Hi-control*). The table also provides expressions that define the causes of value (*Low-Hi*) and timing (*Early-Late*) failures at the output of the valve. From such expressions describing local failure behaviour, in HiP-HOPS it is possible to automatically generate a set of system fault trees and an FMEA which show how component failures cause system level effects.

In the analysis of Fig.1, there is an implicit assumption that point b is always the output of the valve which may be true in a particular system configuration but not in general. To account for flows in the opposite direction, we also need to consider point a as an output, which in practice means that the table has to be extended to include deviations of point a. The symmetry in the design of the valve means that mechanical replication is only required to complete this specification of how the valve behaves in conditions of failure. This specification 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 we expect the component to perform in every application (valve is normally closed unless the value of control signal is 1). For these reasons, the specification of Fig.1 provides a template that could be re-used in different models and contexts of operation, perhaps with some modifications, e.g. on failure rates, to reflect a different environment. We must stress, though, that this type of reuse is only possible because the valve has a small number of well-defined interface points and performs the same simple operation in any context of use.

Generalising this discussion, we can say that the failure annotations of a component can be directly re-used in the same application as long as design changes do not influence the function of the component, i.e. the component receives the same inputs and performs the same operations on these inputs producing the same outputs. Reuse of failure annotations is likely to be possible across different applications for simple components like sensors and simple actuators. On the other hand, the failure annotations for programmable components or components with variable functional profiles will generally have to be re-constructed each time the component is used in a different application with reference to the functions performed in the context of this application. There are, however, exceptions to this rule and opportunities for reuse of safety analyses even for complex components.

PATTERNS OF FAILURE BEHAVIOUR

Complex components, for example components that perform fault tolerant functions, are often designed to provide the same standard failure behaviour on all outputs independently from context of application and the number or type of inputs and outputs [15,16]. A good example of this is the TTP communication controller, a component that handles the communication between a host and other controllers in a time-triggered network. Analysis of the published specification of this component shows that the controller has several important safety properties that should hold in any context of operation [13,14]. These properties collectively define a generic pattern of controller behaviour in the failure domain.

The pattern is schematically illustrated in Fig.2, and shows that the controller is extremely efficient in terms of handling the various classes of failure that analysts would typically examine in a HiP-HOPS study. *Early*, *late* and *commission* failures caused by internal controller faults or the local host are detected and therefore cannot cross the controller-bus interface. Such failures are, therefore, obsolete within the sphere of a TTP network. One reason is that the controller contains mechanisms that prevent the generation of such types of failures. In addition, the controller detects *commission* and *timing* failures generated by the host and transforms these failures into *omissions*. Thus, the only failures that cross the bus interface and enter the sphere of a TTP network are *omission* and *value* failures generated by the host or at the CNI (memory area where host and controller exchange information). Two more types of failure can be generated within the sphere of communications: *omission* and *value* failures caused by external

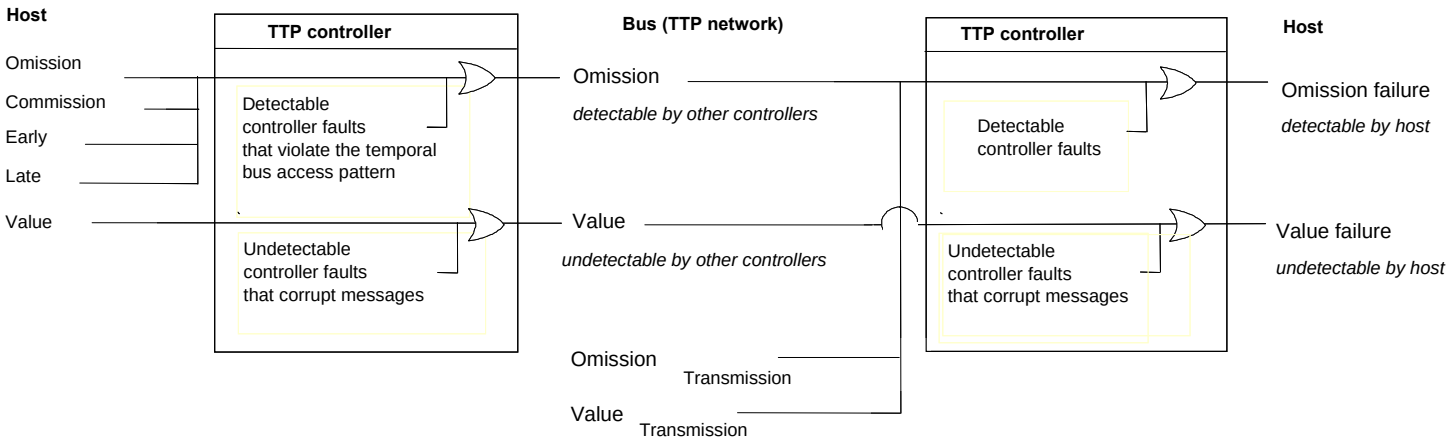


Fig.2. Pattern of failure behaviour of TTP controller (transformation and propagation of failures)

disturbances during transmission (e.g. Electro-Magnetic Interference). From those four classes of failure only one can propagate through and exit undetectable from the TTP network: *value* failures generated by the host or at the CNI.

The pattern of Fig.2 defines an informally specified, convenient, generic abstraction of the controller behaviour in the failure domain. Assuming knowledge of the messages handled by the controller in a given application, this informal pattern could be translated into specific failure annotations described in the language of HiP-HOPS that could then be used together with fault tree and FMEA synthesis algorithms to simplify the safety assessment of TTP networks[1]. Each of these annotations would relate a specific deviation of a message handled by the controller to its possible causes, either a controller failure or a deviation of the message at the input of the controller. Such useful informal use of patterns could be achieved with no extensions to the method. However, informal use implies a manual translation of the informal general pattern to its specific instances within a particular model. We believe that this manual step could be avoided and more effective reuse of failure patterns could be achieved by enabling the formal description of such patterns and their automated adaptation and application in different contexts of design using appropriate tool support.

To achieve this goal, we have extended the language currently used in HiP-HOPS with primitives that can be used to make general statements about failure behaviour. In this language, for example, it is possible to state that "the failure of a component causes omission of *all* outputs of the component" or that "*any* output will be omitted if there is a timing or commission failure of the corresponding input". Such general statements have in the past not been possible in HiP-HOPS (only direct relationships between individual output failures and their explicitly stated specific causes could be stated). However, with the linguistic concepts described in the following section, component failure patterns such as the one illustrated in Fig.2 can now be formally stated. The advantage of this approach is that failure patterns can be mechanically interpreted and, in the context of HiP-HOPS, reused across applications as building blocks for the automatic synthesis of system safety analyses.

LANGUAGE FOR FAILURE PATTERNS

A language for the description of failure patterns should be generic and abstract enough to be technology independent and applicable on systems that integrate diverse technologies (i.e. electrical, mechanical, electronic, software). The definition of such a language must be based on a suitable abstraction of a component in the domain of failure, one that captures all aspects in which failure behaviour is exhibited, but also one that is generic enough to be shared by all components.

In general, all components operate on a set of *inputs* and deliver a set of *outputs*. Inputs and outputs are important in the domain of failure because they form the interface to the external world through which the effects of failure propagate. Components suffer *internal malfunctions* such as electrical or mechanical failures caused by wear or environmental conditions for which components are not qualified (e.g. humidity, temperature, pressure, electro-magnetic interference etc.). Hardware failures follow probabilistic distributions defined by a constant or variable failure rate. Component malfunctions cause local effects on outputs, or *outputs deviations* which, in turn, cause further effects as they propagate through material, energy or data connections to other components. The effects of failure naturally vary depending on failure condition and type of output. However, despite

[1] It is implied that the assumptions about the properties of the communication controller that underlie the pattern of Fig.2 (stated as requirements in its published specification) must be verified on any actual implementation of the controller before the model can be safely used for all the practical purposes of application safety analysis.

differences, all effects fall in a small number of failure classes that are universal and applicable to material, energy and data parameters. These failure classes represent extreme deviations from normal behaviour such as the *omission* or *commission* of outputs or qualitative deviations from correct value (i.e. *hi-low*) and expected timing behaviour (i.e. *early-late*). This classification of failure effects adopts a functional viewpoint which is independent of technology and therefore provides a common basis for describing the local effects of component malfunctions. However, the use of failure classes does not in itself lead to sufficient specifications of failure behaviour, since the role of a component in the domain of failure cannot be fully described by simply enumerating the local effects of its internal failure mechanisms.

Components do not only cause failures, but they also detect and respond to failures caused by other components, the effects of which can be sensed as deviations of inputs. A three way valve with sensing abilities, for example, may detect the absence of flow in one input, and in response, it may automatically restore output flow by switching to an alternative source. On the other hand, in similar circumstances a two way valve will propagate any disturbance of input flow to its output. Although the above cases represent very common patterns of behaviour, components do not only *generate*, *mitigate* or *propagate* failures. They may also *transform* input failures to different types of output failure. An example of such a component is a controller which in response to detected sensor failures omits any further output to ensure that hazardous control action is avoided. This clearly represents a case in which *value failures* at the input are intentionally being transformed into *omission* failures at the output.

To capture those different aspects of local failure behaviour, in HIP-HOPS, logical expressions are used to relate the potential output deviations of a component to a logical combination of causes that include internal malfunctions of the component and deviations of component inputs. Such expressions can be parenthesised and include conjunction, disjunction and negation operators (i.e. *and*, *or*, *not*). An example specification that encompasses different aspects of failure behaviour has been given in Fig.1. The annotations of the two way valve define that the valve propagates all possible deviations of input flow to the output. Value failures of the control signal (*low* and *hi*) are transformed by the valve to course provision failures at the output (i.e. *omission* and *commission* of flow respectively). In addition a number of malfunctions of the valve cause different effects on the outputs. These effects are aggregated in six classes of failure (*omission*, *commission*, etc).

In the example of Fig.1, appropriate keywords have been used to signify o*mission, commission, low, hi, early* and *late* failures in ports *a* and *b*. In general, though, the failure classes considered in the course of the analysis (i.e. *omission*, *commission*, etc) and the identifiers[2] used to describe them are not predefined or strictly prescribed in the context of the proposed language. Analysts have the freedom to define and examine different types of deviations as long as these types are used consistently (i.e. identical types are referenced across the two ends of each connection in the model)[3]. This gives a degree of flexibility which is appropriate for analysis of different types of input/output parameters. For events, i.e. momentary stimuli and responses, for example, only *omission* and *commission* failures could be examined since they sufficiently cover the two possible deviations from nominal state. Similarly for continuous binary control signals, two types of value failure (*hi* and *low*) sufficiently cover all possible deviations from the nominal state of the signal. In this flexible scheme of using failure classes, analysts also have the option of performing very detailed analysis by specifying precise failure conditions such as 'output value>100' or 'value being late by 10msec'. For this to work, though, it is imperative that all such conditions specified in the failure logic of one component have also been considered in the failure logic of connected components.

To summarise the above discussion, in the proposed language, a component is also defined by a vector of failure classes which are applicable to its inputs and outputs. In a given application, a single vector may be shared by all components, but each component may also define its own vector as long as this vector is consistent with the vectors of adjacent components.

One difficulty in the proposed approach is caused by components that handle a large or variable number of inputs and outputs. Take, for example, a communication bus that carries large numbers of different messages in different applications. An electrical failure of the bus, such as a short circuit, will always cause an omission of all outputs. In addition, any deviation of an input message will cause an identical deviation of the corresponding output, unless of course the bus incorporates failure detection and control mechanisms. Although the bus may carry different messages, the above statements provide a pattern of its failure behaviour which is independent of the number and type of those messages. However, the linguistic scheme that we presented so far does not permit the representation of this pattern. In this scheme, each time the bus is used in a different application, analysts must enumerate one by one all potential effects of failure on all output messages and their respectively causes. This is both cumbersome and unnecessary and can be avoided with the introduction of new operators that can make appropriate

[2] For example, the identifiers *O, C, Vh, Vl, Te, Tl* could have been used in Fig.1 to signify the failure classes examined.

[3] Assume, for example, that during the local analysis of component *A*, analysts specify the *omission* of output *out* as possible deviation (e.g. *O-out*). If *out* is connected to input *in* of component *B*, then references in the failure logic of *B* to an *omission* of *in* should use the same identifier (i.e. the event should be specified as *O-in*).

collective references to inputs, outputs, failure modes and failure classes. To enable such collective references, we assume that in the general case each component has a number of vectors of parameters which are important from a point of view of safety analysis: input ports *IP[]*, output ports *OP[]*, Failure Modes *FM[]* and Failure classes *FC[]*. In turn, each input and output port may carry a vector of parameters *P[]*. If, for example:

OP[] = [a,b]
OP[1].P[] = [temp,pres] and
FC[] = [Omission,Commission,Hi,Low,Early,Late]

then *Omission-a.temp* and *FC[1]-OP[1].P[1]* are equivalent references to an *omission* of parameter *temp* in output port *a.* Making references to indexed elements of the vectors, as above, does not really add anything in the declaration of deviations. The introduction of vectors, however, enables us to define four operators that make collective references to the contents of these vectors (even when these contents are yet unknown, vary across applications or simply have not been explicitly stated). The four new operators are namely:

any, *every, majority* and *except*

When the operator *any* is applied on a vector referenced in a failure pattern, it can be substituted by any of the parameters contained in the vector to create multiple instances of the pattern, when in the context of a specific application the parameters of the vector become known. On the other hand, when the operator *every* is applied on a vector, a single instance of the pattern is only created via iteration of all parameters contained in the vector. *Every* can be used in conjunction with logical operators (*and* & or) to create a conjunction or disjunction among all input or output deviations generated by application of the operator.

The *except* operator is used in conjunction with *any* or *every* operators and restricts the global scope of these operators by excluding a set of specified inputs, outputs and failure classes that do not share the common behaviour. Finally, when the *majority* operator is applied on a set of events which are explicitly specified or generated by *every* operators, it causes enumeration of all majority combinations of these events (all 2 out of 3, 3 out of 5, etc., combinations). The effect of the operator is the generation of a logical expression in which events within each combination are conjoint and combinations are disjoint.

In the remainder of the section, we present a catalogue of useful generalized statements about failure behaviour that can be made using the linguistic constructs that we have introduced in this section. For each statement, we provide an informal explanation of its meaning and suggest potential application in the development of patterns for components.

1. Propagation of failure from inputs to outputs

any FC[x] – any OP [y].any P[z] = FC[x]-IP[y].P[z]

> Any deviation *x* of any parameter *z* on output *y* is caused by the same deviation *x* of the corresponding input parameter *z* on input *y*.

Statement can be used for modelling components that directly propagate input failures to outputs, e.g. a communication bus.

2. Global failure in the same mode, e.g. Omission

Omission - any OP[y].any P[z] = FM[1]

> An *omission* of all output parameters is caused by a failure mode of the component.

Statement can be used for modelling of components in which a single failure mode causes the same effect on all output parameters. *Omission* is a typical effect observed in practice when components fail but other common failure effects are also possible, e.g. all parameters produced late.

3. Fail silence & Transformation of failure

Omission - any OP[y].any P[z] =
or (every FC[]-every IP[].P[z])

> *Omission* of all output parameters is caused by any failure of any input parameter

Statement can be used for modelling of components that detect input failures and in response fail silent, i.e. transform *value*, *timing* and *commission* failures into *omissions*. The statement implies very strong failure detection but can be moderated to reflect more realistic fail silent behaviour. The assumption that every failure class at the input is detectable can be modified to exclude *value* failures that are difficult to detect. In this case the statement becomes:

Omission - any OP[y].any P[z] =
or (every FC[] except {Value}-every IP[].P[z])

The statement can also be modified to show other types of failure transformation, e.g. of *timing* failures to *commissions*.

4. Standby - recovery

any FC[x] – OP[1].any P[z] =
Omission-IP[1].P[z] and FC[x]-IP[2].P[z]

> Any deviation *x* of any parameter *z* on the single output port is caused by *omission* of the corresponding input parameter *z* on the first input followed by the same deviation *x* of the corresponding input parameter *z* on the second input.

Statement can be used for modelling of a standby component that monitors a primary component on *IP[1]* and, when an omission is detected, it takes over to continue the provision of the intended function. In this case, any failure on *IP[2]* is propagated to the output of the standby. However, the statement can be modified as shown below to show a fail silent behaviour instead.

Omission FC[x] – OP[1].any P[z] =
Omission-IP[1].P[z] and (FC[x]-IP[2].P[z] or FM[1])

In this case, the standby fails silent in response to either an internal failure mode or any deviation of *IP[2]* that follows *omission* of *IP[1]*.

5. Redundancy on inputs

any FC[x] – any OP [y].any P[z] =
and (FC[x]-every IP[y].P[z])

> Any failure *x* of any parameter *z* on output *y* is caused by a conjunction of the same failure *x* on the corresponding input parameter *z* of every input *y.*

Statement can be used in the modelling of components that rely on redundant inputs for producing correct output as long as one of the redundant inputs is correct.

6. Voting

Omission-OP[1].any P[z] =
majority(every FC[]-every IP[].P[z])

> *Omission* of any parameter *z* on the single output port is caused by a majority of input deviations of any type of the corresponding input parameter *z* on respective inputs.

Statement can be used in the modelling of a fault tolerant component that outputs the majority of inputs that are in agreement. The component fails silent when the majority of inputs have either been omitted, provided at the wrong time, or disagree in value.

FAILURE PATTERNS FOR BUS AND TTP CONTROLLER

The catalogue of failure behaviour presented in the preceding section is by no means exclusive and is intended to only provide guidance for the construction of failure patterns. Using elements specified in this catalogue, it is indeed possible to construct example failure patterns for two components that we have already discussed in preceding sections: a generic communication bus and the TTP controller.

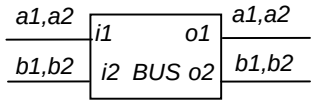
The general failure behaviour of a communication bus combines two elements specified in the given catalogue: propagation of input failures to outputs and global failure in the same mode (in this case, *omission*). Indeed any deviation of an input message sent on the bus will cause an identical deviation of the corresponding output, while failure of the bus causes an omission of all outputs. The two statements below describe this behaviour and together form the failure pattern of a generic bus.

Failure pattern of a communication bus

any FC[x] except {Omission} - any OP [y].any P[z] = FC[x]-IP[y].P[z]
Omission - any OP[y].any P[z] = FM[1]

Given a particular bus, a third statement can be added to link the general bus failure referenced above as *FM[1]* to a particular logical combination of specific bus failures, e.g. *FM[1]=disconnected or shortCircuited.* The pattern can of course be modified or extended to derive more elaborate and detailed behaviours for particular implementations of busses[4].

Fig.3 shows an instance of a bus where two nodes create two communication channels and send parameters *a1*,*a2* and *b1*,*b2* respectively. Other nodes can tap in the corresponding outputs of these two channels, and, via a multiplexer, read some or all messages in a particular order. Assuming that in this context *FC={Omission,Value}* and *FM[1]=busFailed*, application of the bus failure pattern can generate any of the 8 specific output deviations listed in Fig.3 and relate these to causes in terms of bus failure and similar deviations of inputs. The fact that specific failure expressions can be automatically derived from general patterns of failure behaviour once the application context is known is an important one. It means that the fault tree and FMEA synthesis algorithms of HiP-HOPS can interpret patterns into specific failure expressions and, thus, trace the propagation of failures through components that have been described with those patterns.



Omission-o1.a1=Omission-i1.a1 or busFailed
Value-o1.a1=Value-i1.a1
Omission-o1.a2=Omission-i1.a2 or busFailed
Value-o1.a2=Value-i1.a2
Omission-o2.b1=Omission-i2.b1 or busFailed
Value-o2.b1=Value-i2.b1
Omission-o2.b2=Omission-i2.b2 or busFailed
Value-o2.b2=Value-i2.b2

Fig.3. Example bus with two I/O channels and specific deviations derived by application of the bus pattern

[4] We currently consider the development of concepts to enable inheritance and polymorphism in a more structured and familial representation of patterns

Turning now to the failure pattern of the TTP controller, we observe that, once more, the controller combines two elements that we have seen in the catalogue of failure behaviours presented in the preceding section. These are:

- propagation of undetectable value failures of the host (catalogue element #1), and
- fail silence in cases of internal failure, timing and commission failures of the host, and detectable value failures caused during transmission (catalogue element #3).

The controller can assume a dual role as a sender or receiver of messages. The following two failure patterns precisely correspond to these two roles:

Failure pattern of TTP as sender

Omission-any OP[y].any P[z] =
or(every FC[] except {Value}-IP[y].P[z]) or FM[1]

Value-any OP[y].any P[z] = Value-IP[y].P[z]

Failure pattern of TTP as receiver

Omission-any OP[y].any P[z] =
Omission-IP[y].P[z] or Value(detectable)-IP[y].P[z]
or FM[1]

Value-any OP[y].any P[z] = Value-IP[y].P[z]

Note that failures of the TTP controller are collectively represented as a single failure mode *FM[1]* which causes the controller to fail silent. A new class of value failures, *Value(detectable)*, is also introduced to accurately specify the failure behaviour of the controller. The "sender" and "receiver" patterns show that value failures of the host propagate undetected through the TTP network and re-appear as value failures at the output of the receiver. However, the "receiver" pattern shows that another class of value failures, those caused during transmission, are detectable by the receiver and result to omission of corrupted messages.

We believe that patterns such as those presented in this section, could be developed for libraries of components in order to capture the experience generated in the course of difficult and expensive safety studies. In the context of HiP-HOPS, such patterns could be stored in electronic libraries and then reused, either directly or following necessary adaptation, in order to rationalize and simplify the assessment of complex systems.

TOOL SUPPORT

HiP-HOPS is currently supported by a tool that generates fault trees and FMEAs from models developed in SimulationX [17] or Matlab Simulink [18]. These are both mature and widely used engineering tools which provide open architectures and have enabled implementation of the techniques described in this paper[5]. The automated safety analysis tool is experimental but mature and has so far been independently used in complex case studies by Volvo, Germanisher Lloyd and others. A prototypical version of the tool also exists which supports the specification and use of abstract failure patterns.

The architecture of the tool is illustrated in Fig.4. The tool provides a Graphical User Interface (GUI) that enables annotation of components in the model with the failure expressions and failure patterns required for the fault tree and FMEA synthesis. These data become part of the model and are automatically saved and retrieved by the modelling tool (SimulationX or Simulink) every time the model is opened or closed by a user. Failure annotations, including patterns, can be stored in component libraries and be re-used either directly or following modifications within the same model or across different models with the obvious benefit of simplifying the manual part of the analysis.

Once a model has been annotated, the structure of the model and its annotations are saved in a text file. The second component of the FMEA tool is a parser that interprets such files, and reconstructs the enclosed annotated models for the purposes of fault tree synthesis. The synthesis itself is performed by the third component of the tool, the fault tree synthesis algorithm. To generate fault trees, the algorithm performs a backward traversal from each output of the model, in the course of which it evaluates the expressions and patterns contained in the local analyses of the components encountered during the traversal. The resultant network of fault trees is then logically reduced into minimal cut sets. Finally, an FMEA synthesis algorithm operates on these cut sets, and in a single traversal of the cut sets generates a multiple failure mode FMEA.

Note that in a classical manual FMEA only the effects of single failures are typically assessed. One advantage of generating an FMEA from fault trees is that fault trees record the effects of combinations of component failures and this useful information can also be transferred into the FMEA. To accommodate this additional information, the resultant FMEA tables are split into two, one containing the direct effects on the system, i.e. those effects caused by single component failures, and the other containing further effects, i.e. those effects caused by two or more component failure modes. This allows separate, easy access to the most critical information, the single points of failure. Perhaps more importantly, the FMEA shows all functional effects that a particular

[5] It should be noted, however, that the applicability of the proposed technique is not restricted to models developed in these tools. Any model that provides the topology of the system, i.e. components and connections, is suitable for this type of analysis.

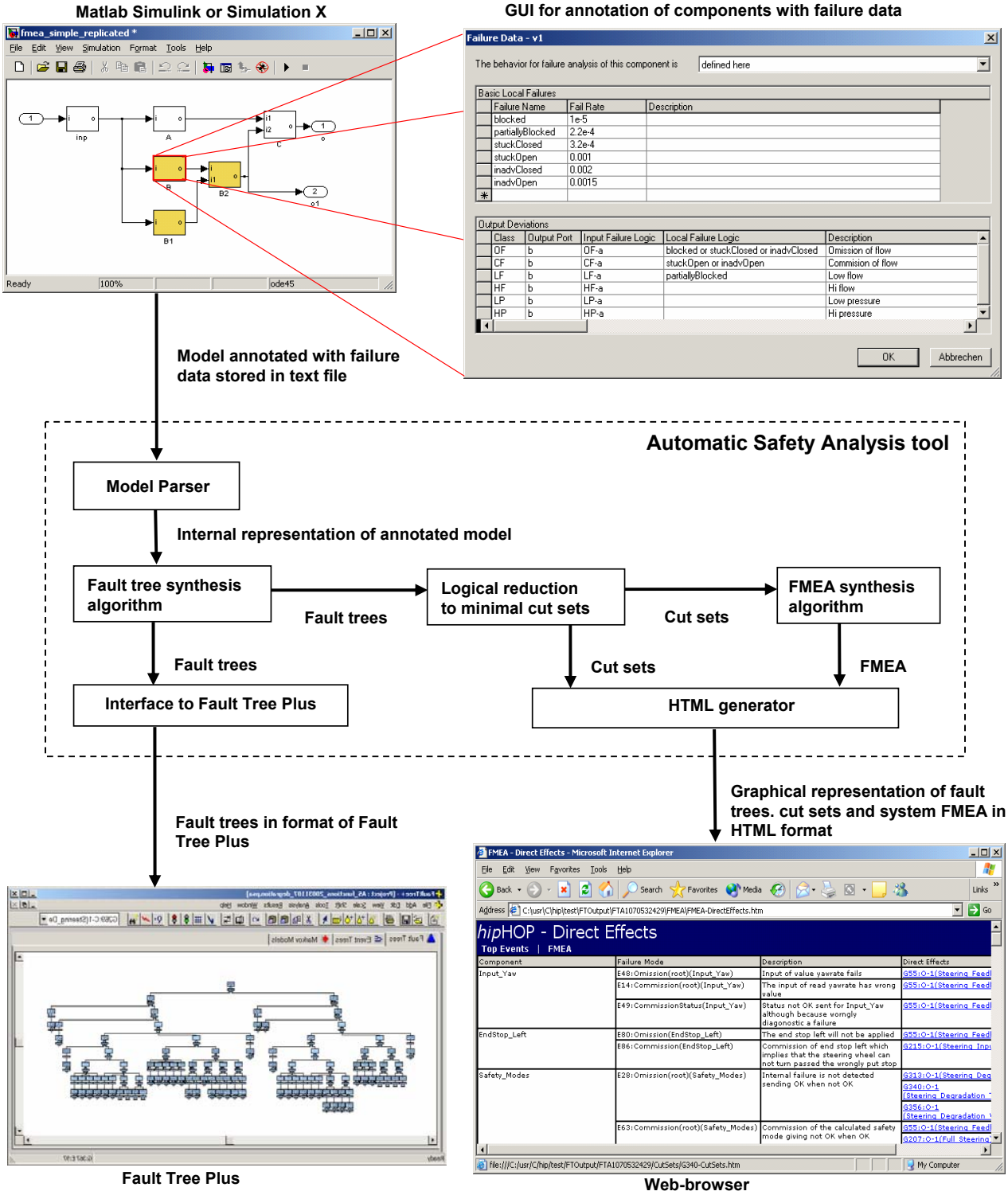


Fig.4. Architecture of the automated safety analysis tool

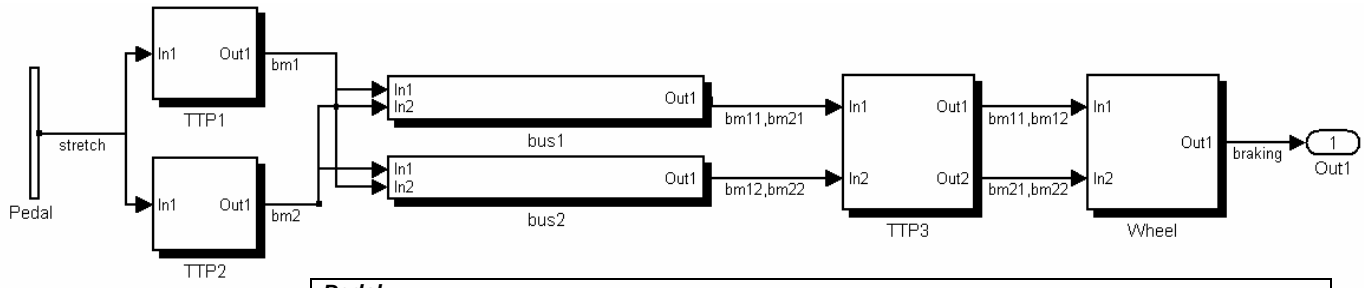
component failure mode causes. The latter is particularly useful as a failure mode that contributes to multiple system failures is potentially more significant than those that only cause a single top event. Precisely because it records the effects of combinations of component failures, this type of FMEA can, in practice, help analysts not only to locate problems in the design, but also to determine the level of fault tolerance in the system, i.e. to determine whether the system can tolerate any single or any combination of two, three or more component failures.

The synthesised fault trees, their analyses and FMEAs are presented in interactive graphical and tabular formats in an HTML viewer. Synthesised fault trees are also exported to Fault Tree Plus (FT+) [19], a widely used fault tree analysis tool, and can be further processed in that tool.

The speed and performance of the automated safety analysis tool are clearly two key factors that will determine the applicability and industrial acceptance of this approach. The proposed fault tree synthesis process is of linear complexity and therefore scales up well with increasing complexity. Large trees are generated by the tool in very short time, which is currently measured in the order of milliseconds for models that contain hundreds of components and thousands of component failures. On the other hand, the synthesis of FMEAs requires calculation of cut sets, a computationally expensive operation where traditional cut set calculation algorithms [20,21] do not scale up well in very large systems. To address this problem, we have developed an implementation of a recently proposed, efficient minimal cut set calculation algorithm [22] which pre-processes fault trees, converting them into BDDs. Improvements in efficiency achieved by this algorithm ensure the scalability of the automated FMEA and will enable, we hope, effective application of the proposed technique in problems of industrial scale. Experimental applications of the tool so far indicate that this approach can indeed lead to fast and efficient ways of generating useful safety analyses from design representations. It generally takes between a few seconds and a few minutes to generate fault trees and FMEAs from models that contain hundreds of components and failure logic that is equivalent to hundred thousands of cut sets. To the best of our knowledge, this performance compares favourably with other results reported in the literature of automated FMEA.

EXAMPLE AND CASE STUDIES

In this section we present an example which demonstrates the use of component failure patterns in the context of HiP-HOPS. Fig.5 illustrates part of a simplified distributed car braking system in which a pedal node communicates the braking demand to a wheel node via a network of TTP nodes connected on a replicated bus. For redundancy, the braking message is replicated and 4 messages are sent from the pedal through the bus to the wheel node. Fig.5 shows the failure annotations of components in this system. These include the patterns of the TTP controller and communication bus that we have already discussed, and annotations for the wheel and pedal nodes. It can be noticed that the wheel node has a single failure mode, *wheelFailed*, that causes omission of braking. The same effect is also caused by omission of all 4 braking messages. On the other hand, a value failure at the output of the system, i.e. incorrect braking, is caused by a common value failure manifested in all 4 messages received by the wheel. The pedal has two failure modes: *pedalFailed* causes omission of the braking message, while *pedalBiased* distorts its value. Note that the linguistic features that we introduced in this paper have been used to produce abbreviated descriptions of failure behaviour for all components. It can also be noticed that the pattern of the communication bus has been modified to include the possibility of detectable value failures caused by electromagnetic interference (*EMI*), a condition considered possible in this environment.



Pedal
Omission-any OP[x].any P[y]=pedalFailed
Value-any OP[x].any P[y]=pedalBiased
Bus
any FC[x] except {Omission, Value(detectable)}-any OP [y].any P[z=FC[x]-IP[y].P[z]
Omission-any OP[y].any P[z] = Omission-IP[y].P[z} or busFailed
Value(detectable)-any OP[y].any P[z]=EMI
TTA as sender
Omission-any OP[y].any P[z] = or(every FC[] except {Value}-IP[y].P[z]) or controllerFailed
Value-any OP[y].any P[z] = Value-IP[y].P[z]
TTA as receiver
Omission-any OP[y].any P[z] = Omission-IP[y].P[z] or Value(detectable)-IP[y].P[z] or controllerFailed
Value-any OP[y].any P[z] = Value-IP[y].P[z]
Wheel
Omission-any OP[x].any P[y]= wheelFailed or and (Omission-every IP[x].every P[y])
Value-any OP[x].any P[y]= and (Value-every IP[x].every P[y])

Fig.5. Model of the braking system and component failure annotations

Given the model and failure patterns of Fig.5, HiP-HOPS generates two fault trees for the events of omission and incorrect braking, and a system FMEA where these two conditions represent the direct or further effects of component failure modes. The fault trees and two FMEA tables are presented in the Appendix in Figures A.1-A.4. We focus on the FMEA as it provides a useful summary of the results of the automated analysis. The direct effects table in Fig.A.3 shows that only 3 component failure modes can cause an omission of braking: *wheelFailed*, *pedalFailed*, and *controllerFailed(TTP3),* i.e. a failure of the TTP receiver. Only one failure mode of the pedal *(biasedPedal*) propagates through the system causes incorrect braking*.* These 4 failure modes represent the only single points of failure and point to areas where the design could be improved. Replication of the pedal node is an obvious such improvement that removes 2 out the 4 single points of failure. In addition to critical failures revealed in the direct effects FMEA table, the further effects FMEA table of Fig. A.4. shows combinations of failures of the two TTP senders and the two busses that also cause omission of braking. Note that since qualitative data about failure modes and their logical relationships have been given, only qualitative analyses are returned (i.e. cut sets and system FMEA). If failure rates have been available they would have been incorporated in the synthesized fault trees and used for calculation of system reliability. The example demonstrates, we hope, the potential for generating relatively complex and useful analyses with relative ease. The same techniques are currently applied on much larger case studies which include a steer-by-wire prototype system designed by Volvo cars.

In the context of this case study, a functional model was first developed in Matlab Simulink and this was deliberately designed without any degraded or fallback modes, in order to test whether the analysis could help in the systematic identification and design of such modes. The model was annotated with local analyses and failure patterns and then fault trees and FMEAs were automatically generated in several iterations of this process. An indication of the complexity of the model is that the analysis results to tens of thousands of cut sets. However, it takes less than a second in an average personal computer to generate and evaluate the fault trees and FMEAs.

Omission and *course value* failures were considered in the course of the analysis. Fault trees and FMEAs therefore show how omissions and value failures of input, processing and actuator functions cause system level effects, i.e. omissions or value failures of steering functions. The manual assessment of the severity of those effects helped to identify the criticality of causes (i.e. failures of input, processing and actuator functions) and this in turn assisted the design of these basic functions. For example, wherever the analysis indicated that the omission of a function had only marginal effects while a course value failure had catastrophic effects, a design recommendation was made to design the function in a way that it fails silent. This in turn led to the identification of several degraded modes in which non-critical steer-by-wire functions may fail silent with only marginal effects on the system. A state-chart was then constructed to show how graceful transition to such modes could be achieved. It would be impossible to describe in the space provided the voluminous results of this study. However, to highlight the practical value of the analysis, in Fig. A.5 we present the high level state-chart that was derived as a result of the straightforward interpretation of these results. The chart of Fig. A.5 shows how a number of “critical” (as indicated by the analysis) failures of input, processing and actuator functions should lead the system safely into assisted-mechanical and progressively unassisted steering modes. Within the main “steer-by-wire” mode it is also possible to see how a number of “less critical” (as indicated by the analysis) functional failures should lead the system into sub-modes where some of the steer-by-wire functionality is lost but the system can safely remain in the normal “steer-by-wire” mode.

CONCLUSIONS

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 new safety analysis techniques that can address these difficulties by exploiting reusable component based specifications of failure, or context independent failure patterns. We also proposed a linguistic concept to enable representation and reuse of such patterns in the context of HiP-HOPS, a recently proposed technique for compositional safety analysis. Furthermore, we demonstrated a set of useful patterns including one for the TTP communications controller and demonstrated their use in a small example derived from an automotive system.

Using the concepts presented in this paper, component failure patterns could be developed for libraries of components in order to capture the experience generated in difficult and expensive safety studies. Such patterns could be stored in electronic libraries and then reused, in the context of HiP-HOPS, in order to rationalize and simplify the assessment of complex systems. An experimental but mature tool supports the proposed process and can be made available for independent application of this approach. The tool has so far dealt with design problems of medium complexity in which annotated components are in the order of hundreds. Problems may arise, though, in large systems that contain thousands of annotated components and may result in a failure logic that is composed of millions of cut sets. One way of simplifying the analysis in such cases is by structuring the model as a hierarchy of subsystems. At low levels, the model may still incorporate thousands of components, but the annotation can now be performed at a higher level of abstraction in the hierarchy where there are a smaller number of components or subsystems to annotate.

Potential benefits from application of the proposed approach in large scale are substantial and include

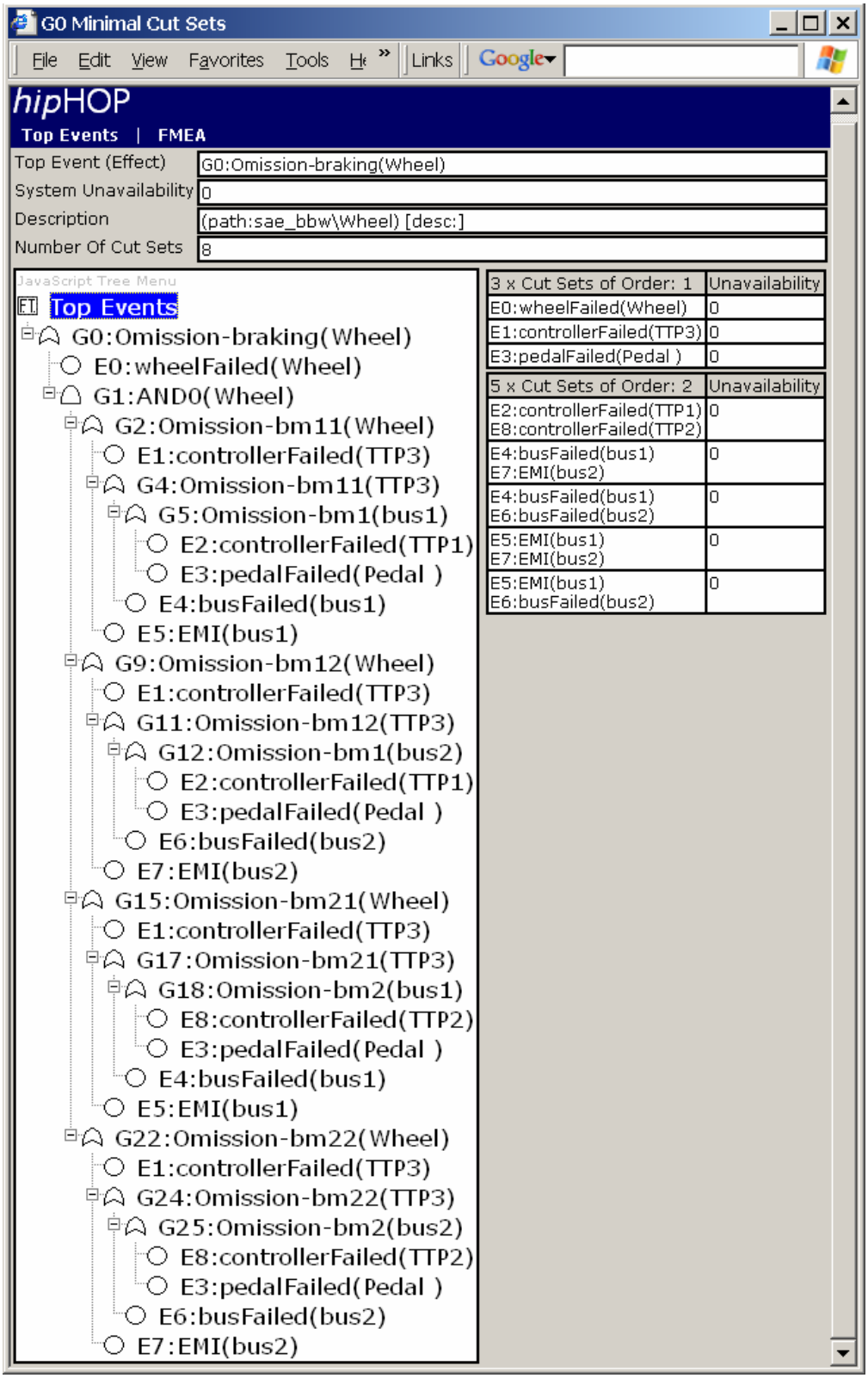
easing the examination of effects of design modifications on safety and keeping the safety analyses consistent with the design.

One area of further work is extension to enable temporal safety analysis. Hitherto application of HiP-HOPS has produced classical combinatorial fault trees which are equivalent to those produced via manual analysis. Although such fault trees are useful for establishing critical combinations of component failures, they miss the temporal ordering of events and cannot explain the significance of this ordering in the failure behaviour of the system. We currently extend HiP-HOPS to enable synthesis and analysis of fault trees that capture temporal relationships between events. The Priority AND (PAND) gate, a long established but vaguely defined component of the fault tree vocabulary [22], and a new Simultaneous AND gate (SAND) are rigorously defined and a set of temporal laws is formed that is used to reduce branches of the tree containing many temporal gates. With a temporal logic in place, fault trees containing PAND and SAND gates can then be qualitatively analysed, and so form a set of ordered minimal cut sets, or 'minimal cut sequences'. We are currently writing up the first results of this work which we hope to publish soon. To enable further integration of safety analysis in the design process, we are also combining this work with recent advances in evolutionary optimisation [23]. The aim of this work is to further automate difficult aspects of system design such as the cost effective allocation of component redundancies and the apportionment of safety and reliability requirements on components of the system during design.

REFERENCES

1. *www.vv.se/aktuellt/pressmed/2003/hkpress19.htm*
2. Ye F. and Kelly T. P., "Criticality Analysis for COTS, Software Components", *22nd International System Safety Conference* (ISSC'04), 2004.
3. Shamus P. Smith and Michael D. Harrison, "Measuring reuse in hazard analysis", *Reliability Engineering & System Safety*, 89(1) 93-104, 2005.
4. CENELEC, "Railway applications: Specification and demonstration of dependability, reliability, maintain-ability and safety", *EN 50126-9*, 2000.
5. Mernick M., Viljem Zumer V., "Reusability of formal specifications in programming language description", *8th Annual Workshop on Software Reuse, WISR8*, Columbus, Ohio, pp. 1 - 4, 1997.
6. Thane H., Wall A, "Formal and Probabilistic Arguments for Component Reuse in Safety-Critical Real Time Systems", *Technical report CBSE – State of the Art*, Mälardalen University, 2000.
7. Fenelon, P., McDermid, J.A., Nicholson, M., Pumfrey, D.J., "Towards Integrated Safety Analysis and Design", *ACM Applied Computing Review*, 1994.
8. Papadopoulos, Y., McDermid, J. A., "Hierarchically Performed Hazard Origin and Propagation Studies", *SAFECOMP '99,* LNCS, 1698 139-152, 1999.
9. Kaiser, B., Liggesmeyer, P., Mäckel, O., "A New Component Concept for Fault Trees", *8th Australian Workshop on Safety Critical Systems and Software (SCS'03)*, Adelaide, 2003.
10. Kaiser, B., Gramlich, C., "State-Event-Fault-Trees: A Safety Analysis Model for Software Controlled Systems", *SAFECOMP'04*, LNCS, 3219:195-209, 2004.
11. Papadopoulos Y., McDermid J. Sasse A. R., Heiner G., "Analysis and Synthesis of the Behaviour of Complex Programmable Electronic Systems in Conditions of Failure", *Reliability Engineering and System Safety*, 71(3):229-247, 2001.
12. Papadopoulos Y., Parker D., Grante C., "A method and tool support for model-based semi-automated Failure Modes and Effects Analysis (FMEA)", *9th Australian Workshop Safety Critical Programmable Systems (SCS'04)*, Brisbane, 2004
13. Kopetz, H., "The time-triggered approach to real-time system design", *Predictably Dependable Computing Systems*, ESPRIT basic research series, Springer-Verlag, Berlin, 1995.
14. Kopetz, H. and Grunsteidl, G., "TTP-A protocol for fault tolerant real-rime systems", *IEEE Computer*, 27(1):14-23. 1994.
15. Fliege I., Geraldy A., Gotzhein R., Kuhn T.and Webel C., "Developing safety-critical real-time systems with SDL design patterns and components", *Computer Networks*, June 2005.
16. Grunske, L., "Transformational Patterns for the Improvement of Safety Properties in Architectural Specification", *2nd Nordic Conf. on Patterns Languages, Norway*, Sept. 2003.
17. ITI, "Simulation X", *www.iti.de*, 2003.
18. Mathworks, "Matlab",*www.mathworks.com*, 2005.
19. Isograph, "Fault Tree +", *www.isograph.com*, 2005.
20. Fussell, J.B., Vesely, W.E., "A new methodology for obtaining cut sets for fault trees", *Transactions of the American Nuclear Society*, 15:262-263, 1972.
21. Pande, P.K., Spector, H.E., Chatterjee, P., "Computerized fault tree analysis: TREEL and MICSUP", *Operations Research Center, University of California, Berkeley, ORC 75-3*, 1975.
22. Sinnamon, R. M., Andrews, J. D., "New approaches to evaluating fault trees", *Reliability Engineering and System Safety*, 58:89-96, 1997.
23. Vesely, W.E., Goldberg, F.F., Roberts, N.H., Haasl, D.F., "Fault Tree Handbook". Washington D.C., USA. *US Nuclear Regulatory Commission*¸1981.
24. Papadopoulos Y., Grante C., "Evolving car designs using model-based automated safety analysis and optimisation techniques", *Journal of Systems and Software*, Elsevier Science, 76(1):77-89, 2005.

APPENDIX



3 x Cut Sets of Order: 1	Unavailability
E0:wheelFailed(Wheel)	0
E1:controllerFailed(TTP3)	0
E3:pedalFailed(Pedal)	0

5 x Cut Sets of Order: 2	Unavailability
E2:controllerFailed(TTP1) E8:controllerFailed(TTP2)	0
E4:busFailed(bus1) E7:EMI(bus2)	0
E4:busFailed(bus1) E6:busFailed(bus2)	0
E5:EMI(bus1) E7:EMI(bus2)	0
E5:EMI(bus1) E6:busFailed(bus2)	0

Fig A.1. Fault tree for Omission of braking.

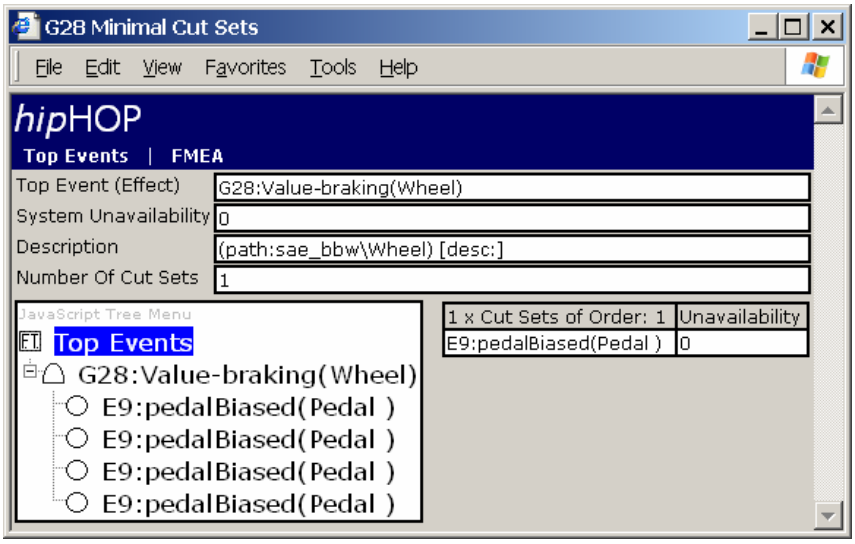
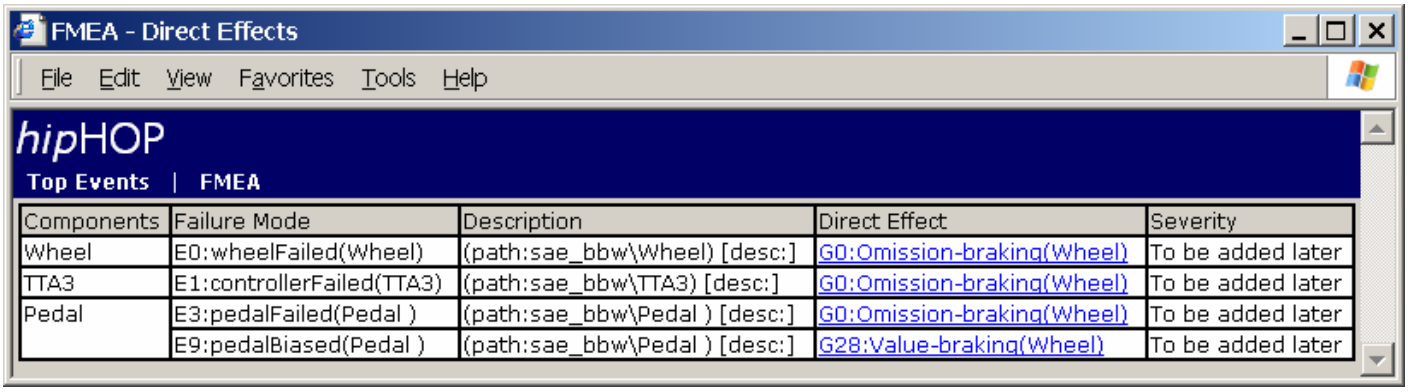
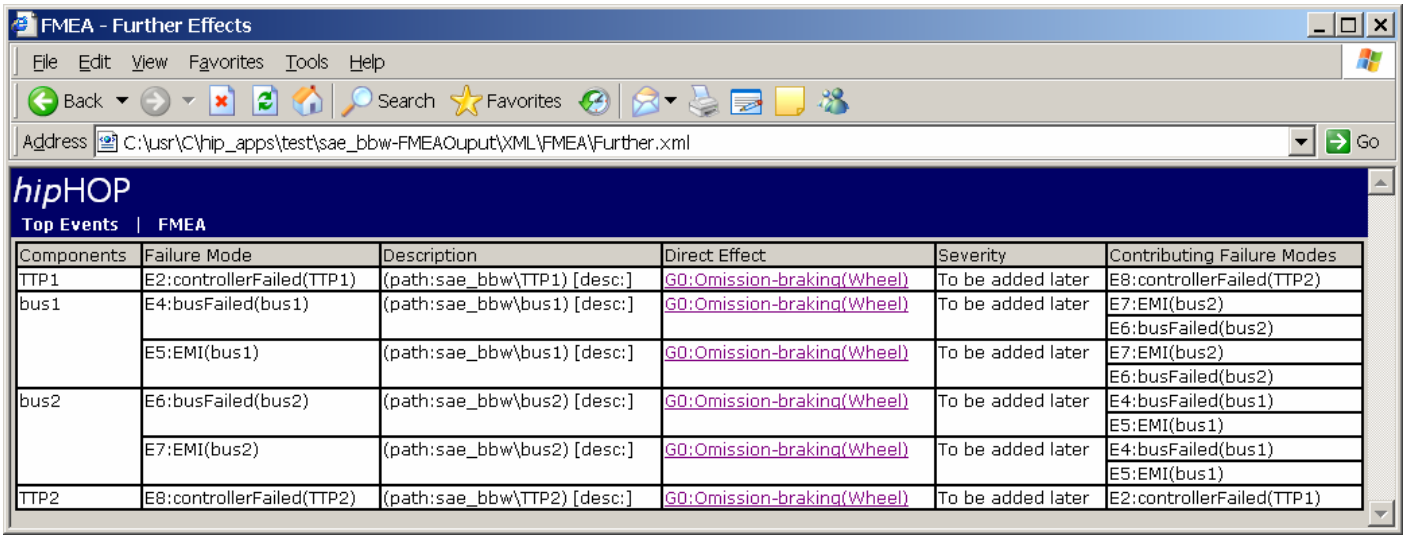


Fig A.2. Fault tree for incorrect braking.



Components	Failure Mode	Description	Direct Effect	Severity
Wheel	E0:wheelFailed(Wheel)	(path:sae_bbw\Wheel) [desc:]	G0:Omission-braking(Wheel)	To be added later
TTA3	E1:controllerFailed(TTA3)	(path:sae_bbw\TTA3) [desc:]	G0:Omission-braking(Wheel)	To be added later
Pedal	E3:pedalFailed(Pedal)	(path:sae_bbw\Pedal) [desc:]	G0:Omission-braking(Wheel)	To be added later
	E9:pedalBiased(Pedal)	(path:sae_bbw\Pedal) [desc:]	G28:Value-braking(Wheel)	To be added later

Fig A.3. Direct effects FMEA.



Components	Failure Mode	Description	Direct Effect	Severity	Contributing Failure Modes
TTP1	E2:controllerFailed(TTP1)	(path:sae_bbw\TTP1) [desc:]	G0:Omission-braking(Wheel)	To be added later	E8:controllerFailed(TTP2)
bus1	E4:busFailed(bus1)	(path:sae_bbw\bus1) [desc:]	G0:Omission-braking(Wheel)	To be added later	E7:EMI(bus2)
					E6:busFailed(bus2)
	E5:EMI(bus1)	(path:sae_bbw\bus1) [desc:]	G0:Omission-braking(Wheel)	To be added later	E7:EMI(bus2)
					E6:busFailed(bus2)
bus2	E6:busFailed(bus2)	(path:sae_bbw\bus2) [desc:]	G0:Omission-braking(Wheel)	To be added later	E4:busFailed(bus1)
					E5:EMI(bus1)
	E7:EMI(bus2)	(path:sae_bbw\bus2) [desc:]	G0:Omission-braking(Wheel)	To be added later	E4:busFailed(bus1)
					E5:EMI(bus1)
TTP2	E8:controllerFailed(TTP2)	(path:sae_bbw\TTP2) [desc:]	G0:Omission-braking(Wheel)	To be added later	E2:controllerFailed(TTP1)

Fig. A.4. Further effects FMEA.

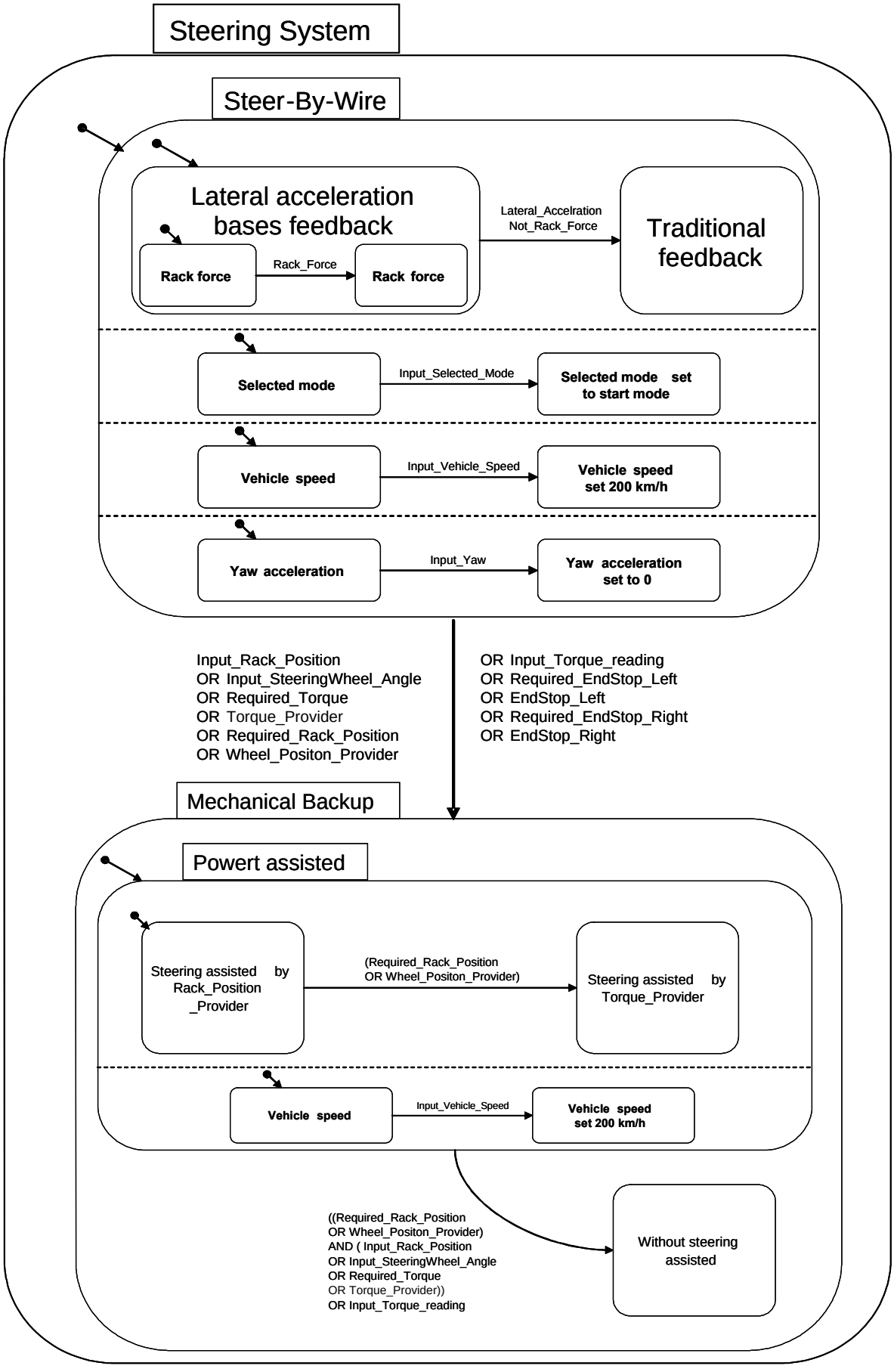


Fig. A.5. Modes of a steer-by-wire prototype derived with the aid of automatically synthesized safety analyses