Failure Modelling in Software Architecture Design for Safety

Weihang Wu Tim Kelly
High Integrity Systems Engineering
Department of Computer Science
University of York
York, UK, YO10 5DD
{wwu, tpk}@cs.york.ac.uk

ABSTRACT
In mission-critical industries, early feedback on the safety properties of a software system is critical and cost effective. This paper presents a compositional method for failure analysis of a system based on the proposed software architecture. This method is based upon the use of CSP as the failure modelling language and its associated tools as failure analysis. Preliminary findings from the application of this approach are also presented.

Categories and Subject Descriptors
D.2.11 [**Software Engineering**]: Software Architectures – *information hiding, languages, patterns.*

General Terms
Design, Languages, Reliability, Verification.

Keywords
Failure Modelling, CSP, Software Architectures, Safety Analysis.

1. INTRODUCTION

1.1 Motivation
Over the last decade, there has been a move to architecture-based development for large, complex software systems in the safety domain. An architecture-based development approach can reduce the impact of change and facilitate effective component reuse. The safety of a software system can also be heavily influenced by architectural decisions. Architectural decisions and the rationale behind them must be well justified and evaluated for their impact on safety. Architectural assumptions, the safety properties of architectural elements and their dependencies must be explicitly identified and documented. However, existing practice fails to systematise solutions to these architectural issues. Examples of the most serious computer-related accidents in the past 20 years, such as Therac-25 [15], and Ariane 5 [17], can be attributed to the flawed system and software architectures, where architectural assumptions regarding safety properties of a reused component were implicit and invalidated. One possible solution is to provide effective feedback to the architecture development process.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
WADS'05, May 17, 2005, St. Louis, MO, USA.
Copyright 2005 ACM 1-59593-124-4/05/0005…$5.00.

The potential benefits of early feedback of the viability of a software system have been well recognised [8, 15] in mission-critical industries: Firstly, it allows exploration of the impact of architectural decisions and discovery of safety problems early enough to avoid rework of system design. Secondly, it enables validation of existing requirements and elicitation of new safety requirements to be addressed the subsequent design refinement process. Thirdly, it facilitates communication between software and hardware development processes and thus hardware/software safety tradeoffs can be explored. From the point of view of safety, such early evaluation predicts not only how a software system behaves in normal conditions but also in the presence of failures.

The work outlined in this paper follows by our previous work on safety tactics [23], in which a software architecture design method for safety was proposed. One key problem we experienced when applying safety tactics is the selection of proper safety-quality scenarios, which will determine the possible safety tactic candidates and thus influence the 'shape' of the architecture. In addition, the protective mechanisms (termed *safety tactics*) employed may themselves fail and introduce further failures, thereby generating new failure scenarios. An effective method to evaluate software architectures for safety is thus needed.

1.2 Software Safety Analysis
The basic tenet of software safety analyses is that software safety is the system property and must be considered as part of the whole system safety [15]. One of the main challenges in software safety analysis is thus how to integrate analyses of software, hardware, and environment failures in an effective manner. The key to the effectiveness of software safety analysis lies in at least the following five aspects: an underlying formal model where imprecision and ambiguity can be avoided; compositional capabilities which determines safety properties of a system by the composition of its components; a systems modelling approach where hardware, software and even human behaviour can be modelled in a unified or integrated model; sufficient expressive power to capture common failure scenarios such as sequential failures and single-point failures in a straightforward manner; automation support to enable safety analysis to be repeated with minimal effort. Software architecture addresses software issues such as flexibility and development costs by providing an abstract model of a system in terms of software elements that have external visible properties and their interactions [6]. Such compositional features of architectures must be explicitly acknowledged in its corresponding safety analysis. Traditional safety analysis techniques [15] such as Fault Tree Analysis (FTA) rely on monolithic decomposition into system modules with regards to the hierarchy of failure influences rather than to the architectures

and thus are inadequate to demonstrate effectiveness at an architectural level, where compositional capability is required. Furthermore, current techniques mainly rely on human capability to analyse and compose safety analysis data and provide little support for automated analysis.

1.3 Objectives of This Work

The goals of this work are (i) to describe failure mechanisms of a software system in a compositional manner, (ii) to integrate architectural design and safety analysis processes, and (iii) to support automated safety assessment. Our method requires an architectural description as input, and transforms it into a CSP system model, and then extends the system model enriched with failure information to form a failure model. The effect of such extension is similar to fault injection [24] and achieves two separate objectives: failure forecasting (to evaluate the effectiveness of protective mechanisms employed), failure removal (to eliminate or mitigate the effects of unacceptable failure scenarios by informing later design decisions).

This approach is discussed in the following sections. Section 2 discusses the use of a formal language Communicating Sequential Processes (CSP) [11], which forms the basis for our failure modelling. Section 3 describes our presented method. Section 4 discusses preliminary findings. Finally, section 5 draws some conclusions and discusses the ongoing work.

2. VALUE OF CSP

CSP was originally devised to solve problems of communicating or concurrent systems (i.e., freedom of deadlocks and livelocks) at an implementation level. Later work on CSP was refined towards formal specification of systems (either software, hardware, or even human) behaviour in terms of patterns of event sequences or component interactions. In recent years CSP has been incorporated into an architecture description language Wright [4] for the purpose of specifying architectural connectors. However, there have been few industrial applications of Wright. We also note that there are two important tools available for CSP: an animator ProBE [3] and a model checker FDR2 [2], both of which complement each other and are essential for automated analysis. The additional possibilities opened up by recent timed and probabilistic extensions to CSP convince us that CSP is a powerful compositional modelling language for hybrid software/hardware systems. Modelling problems such as event sequences [15] and component interactions [20] have been recognised as first-class entities in software safety analysis. We thus believe that CSP offers great benefits to failure modelling.

Another reason to choose CSP is its explicit notation for specifying nondeterminism. Nondeterminism is not a feature of real software systems but is one of specification at a higher level of abstraction due to incomplete knowledge about system behaviour. Similarly, failure behaviour of an architectural component could be uncertain because of lack of implementation details or limited model views or unresolved design decisions. The explicit modelling of uncertainty identifies alternative failure paths in an unconstrained manner.

Extensive work has been conducted towards modelling functional aspects of systems using CSP. Equally important for safety is to know whether it can be applied to model the failure mechanisms of a system. We observe that there is a very close relationship between failure modelling and system modelling. The key difference is that all failure events are *explicitly observable* within the context of failure modelling, whilst in the framework of system modelling, only normative events are observable and failure events are implicitly seen as the *non-occurrences* of normative events. We thus believe that the system modelling languages such as CSP can also be extended to model failure behaviours. The incorporation of both normative models and failure models using the same language can thus form a unified safety model. To make use of CSP tool support, we assume machine-readable version of CSP (CSP_M) in this paper.

3. APPROACH

Our interest is the effective determination and evaluation of failure behaviour of the whole system in terms of composition of its architectural elements. The effectiveness lies in the inherent compositional capability and the availability of tool support. And failure behaviour of each component is modelled in terms of failure propagation and generation. The result of such modelling is the combination of different possible failure flows from external failures or components' internal failures to system-level failures. The remainder of this section presents our approach in respond to the following questions: how can failure modelling assist the architectural design process? How can CSP be applied to failure modelling problems? How can we define failure behaviour of a component? How does a failure model facilitate safety analyses?

3.1 Relationship to Architectural Design

Our method concerns failure mechanisms of a software system by establishing a correspondence with its underlying software architecture. We distinguish three fundamental elements in the definition of an architecture: architectural building blocks such as components and connectors, architectural decisions such as safety tactics which are taken in order to address specific quality attributes, and architectural views that express separation of concerns in design. We assume all the three elements are essential to software architecture description or documentation. In CSP, the basic unit is an event, and the building block is a process representing a pattern of event sequences (i.e., behaviour of a component). Only processes representing failure behaviours of each component are considered in our approach. We also introduce the notion of viewpoints into our modelling framework to isolate different safety concerns such as functional failures and hardware failures. As a result, the correspondence between failure modelling and software architecture should be defined in terms of the links between architecture views and failure model views, and those between architectural building blocks and CSP elements.

We treat architectural design as an iterative and incremental development process from intermediate design models to a complete and rigorous solution. To provide effective feedback to the architecting process, the failure modelling activities should take place as early in architectural design as possible and support incremental and iterative development. Figure 1 illustrates the relationship between failure modelling and architectural design with respect to four activities: (1) architecture transformation, where a system model is generated based on an architecture description, and preliminary feedback can be produced regarding the validity of the system model; (2) failure modelling, in which the system model is extended by taking into account failure behaviours of each component; (3) scenario generation to identify failure scenarios of interest associated with the failure model; (4) safety analysis that produces assessment results and feedback to the subsequent design process.

Figure 1. Failure modelling in architecture design process

3.2 Architectural Transformation

Architecture transformation is a step that identifies the CSP events and processes (i.e., a system model) based on an architecture model. In general, architectural elements such as components, connectors and subsystems are suitable process candidates. The basic tenet of process identification is the fundamental communication principle of CSP – handshaking synchronisation (i.e., synchronised events only occur when both the process and its environment allow them). Interactions between identified processes must follow this principle. Any violation means the need to identify new processes. It should be noted that this step could be applied to a wide range of architectural representations from informal block-line diagrams such as UML [12] to formal descriptions such as AADL [9] provided that proper transformation rules can be defined.

The transformation must distinguish between elementary and composite components such as subsystems and systems in architecture. Elementary components can be modelled and analysed in isolation from the rest of the system, whilst the behaviour of a composite component can be determined by behaviours of enclosing components. Hence, the main focus of failure modelling would be the definition of failure behaviours of elementary components.

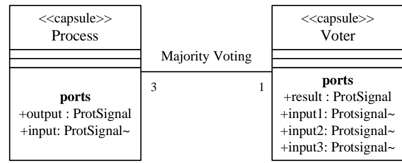
The technique of viewpoints has been widely used in software architecture design. An architectural view presents a specific aspect of a software architecture for the purpose of separation of concerns. The composition of all architectural views can be argued to represent all aspects of a software architecture. Common architectural views discussed and adopted in the software architecture community are [6]: component-and-connector (C&C) view, module view, and allocation view. Both C&C and allocation views are essential for gaining a clear picture of the failure behaviour of a system.

CSP itself does not support the notion of viewpoints or views. However, like other architectural elements, architectural views can be mapped directly onto CSP processes. The incorporation of architectural views into failure modelling can facilitate incremental and iterative modelling, since not all architectural views are safety-related and failure modelling can be performed within one view in isolation from other views. A failure model view gives us a specific perspective on the failure behaviour of the system. Internally generated failure events of a component within one view can be treated as a special class of propagated failure events, in which their external stimulus has been hidden by its container view. And the composition of various views gives a detailed picture on the failure flows between views. For example, the C&C view exhibits functional failure behaviour with underlying hardware details hidden, whilst the hardware view presents the behaviour of random hardware failures. The composition of the two views can model how low-level hardware errors can contribute to high-level functional failures. Hence, our method supports two levels of composition: composition of individual components' failure behaviour within one view, and composition of failure behaviours from different views.

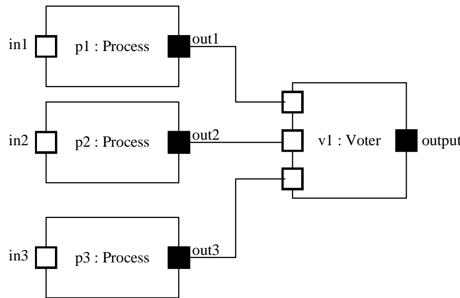
Another important general feature for the description of architectures is styles or families of systems. The basic building block for architectural styles is a type system, in which types of architectural elements such as components and connectors are defined. CSP incorporates powerful functional-language-like constructs for data types and expressions, but no support for types of processes has yet been provided. One possible solution is to treat an architectural type as a special family process – a process factory, creating instances of component process. Both the family process and its instances share the same patterns of failure behaviours. Notably, CSP provides a useful renaming operator to share component behaviours. A typical example would be that both I/O device and memory at a high level of abstraction share the same pattern of failure behaviour. Given that the failure process of memory (MEM_F) has been defined, it can follow that:

IO_F = MEM_F [[memread <- ioread, memwrite <- iowrite]]

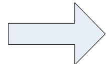
Figure 2 illustrates an example mapping of a C&C view of a triple modular redundancy (TMR) system documented using UML-RT [22] to a CSP model. A collaboration diagram and a class diagram are shown to represent the instance and the type relationships for the TMR system, respectively, within a C&C view. Due to space constraints, we assume familiarity with UML-RT such as capsules, connector and ports. Since there are two types of components introduced in the C&C view, there are two corresponding family processes in the CSP model – PROCESS and VOTER. Component instances are mapped onto new CSP processes given that they share the same patterns of behaviours with their family process. Connectors and ports are mapped onto CSP channels followed by the synchronisation principle. Finally, the C&C view must be mapped onto a composite CSP process, in which its failure behaviour can be determined upon the composition of its enclosing processes.

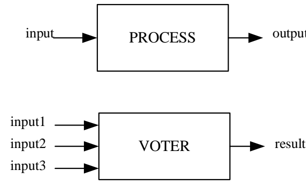


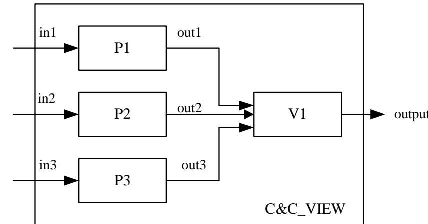
UML-RT class diagram for Majority Voting style



UML-RT collaboration diagram for TMR system







P1 = PROCESS [[input <- in1, output <- out1]]
P2 = PROCESS [[input <- in2, output <- out2]]
P3 = PROCESS [[input <- in3, output <- out3]]
V1 = VOTER [[input1 <- out1, input2 <- out2, input3 <- out3, result <- output]]

CSP model

Figure 2. A TMR example of architecture transformation

3.3 Basic Failure Modelling

For the safety purpose, failure events must be distinguished from normal events in failure modelling. CSP supports the definition of multi-part events by the introduction of infix dot. Hence, our convention rule is that all events must have one component describing normal or failure conditions (such as *sensor.failed* and *processor.working*). Failure modes can be added following the failure description component if necessary (such as *input.failure.omission*. Failure modes can be usually identified and classified using a set of guidewords such as those from SHARD [19]. Upon the availability of failure modes, they can be allocated to the system model (i.e., failure mode allocation).

One key concern in failure modelling is its expressive power with regard to describing failure behaviour of a component. Component failure results from the occurrence of some form of failure stimulus (i.e., a failure event) and subsequent occurrences of different possible normal intended events and failure events (i.e., failure flows). Failure stimuli can arise in the external environment, or inside a software or hardware component of the system. The definition of failure behaviour can be recursive – an internal component failure can propagate to its container component and possibly to the whole system. CSP provides a rich set of operators to capture various forms of these failure flows. Combination of various failure flows can be modelled by making the occurrence of each flow a deterministic choice or nondeterministic choice. The decision regarding the choice between determinism and nondeterminism depends on the degree of the knowledge about the failure behaviour of a component. Nondeterminism reflects uncertainty about the failure flows will occur, whilst deterministic choice removes this uncertainty by making the occurrence of failure flows visible to its environment.

Common failures such as transient failures (that occur once and then disappear), permanent failures (that continue to exist until protective events for them), and arbitrary failures (that exhibit different failure events in a nondeterministic manner) can be modelled by CSP in terms of sequencing and recursion. For example, typical failure behaviours of a processor can be: crash failures (i.e., permanent omission failures), transient timing failures, transient value failures, and corruption failures (i.e., arbitrary timing and value failures). The following CSP process definition illustrates these concepts using the deterministic choice:

```
-- Crash failure
CPU_CH = cpu.failure.omission -> CPU_CH
-- Transient timing failures
CPU_TF = cpu.failure.timing -> CPU_TF
         [] cpu.ok -> CPU_TF
-- Transient value failures
CPU_VF = cpu.failure.value -> CPU_VF
         [] cpu.ok -> CPU_VF
-- Corruption failures
CPU_CRT = CPU_TF [] CPU_VF
```

We distinguish two basic forms of failure flows within a component: failure propagation in which failure stimulus arises in the environment of the component, and failure generation where failure stimulus arises inside the component and is invisible to its environment. Failure generation can be seen as a special class of failure propagation where the cause of the failure stimulus has been hidden by the architectural view. Inconsistency may arise when both failure flows interact at some point. For example, imagine a timing failure that arrives at the input of a component *C*. This timing failure would propagate through *C* if there is no protection mechanism employed in *C*. If *C* itself generates an omission failure during the propagation, then what failure should be produced eventually at the output of *C*? To answer this question involves consideration of the relationship between failure flows (e.g., which flow arrives at output first?), and between failure types (e.g., does one type of failures override another type?). We treat omission and commission failures as two special classes of both timing and value failures and assign them the highest priorities, whilst value and timing failures are

considered of equal priority. As a result, the higher-priority failure events always override the lower ones when they interact.

Special care should be taken when modelling the behaviours of protection mechanisms. In many cases, the manner in which protective devices handle failures should be obvious under normal conditions. However, the situation becomes more complex when considering failures of these protective devices. Consider, the simple scenario if a working watchdog timer employed within a component that can detect omission and late timing failures via its underlying timeout mechanism. What happens if watchdog itself fails? Will the failure stimulus be propagated through the component or transformed into another type? The answer to this scenario may depend on the internal details of the design or implementation, which may not be available at current design stage. From the perspective of safety, it is worthwhile to assume the worst case in failure modelling. We thus propose to specify the occurrences of all possible failure flows introduced by nondeterministic choice upon the failure of a protection device. This assumption can be refined in the following design iterations when more knowledge about the system is obtained.

Figure 3 shows the definition of the failure behaviours of the family processes PROCESS and VOTER of the TMR example. For simplicity, we only assume omission and value failures in this example. The Process component family is responsible for computing the results based on the input data received, and thus can propagate any incoming failure from its input to output ports. It may also generate omission or value failures due to its underlying hardware faults. Comparably, the Voter component family is responsible for choosing the 'correct' result among three redundant input channels, and thus can detect one faulty input channel and stop its failure propagation. But the voting protection mechanism will be disabled if two or more faulty channels agree with each other. The Voter itself can generate omission failures (i.e. fails stop) once it can't make a decision upon voting or its underlying hardware fails.

```
-- we can define a SHARD failure mode
datatype SHARD_FM = O | C | E | L | V

-- we can combine SHARD failure mode and normal mode together
-- or we can just define a generic datatype with only two states (failed or not)
datatype T = failure.{ c | c <- SHARD_FM} | ok

-- we then associate each type with a functional value
f(ok) = 0
f(failure.O) = 1
f(failure.C) = 2
f(failure.E) = 3
f(failure.L) = 4
f(failure.V) = 5

-- we can only be interested in a particular subset of the predefined types
-- T1 = {ok, failure.O, failure.V}
subtype T1 = failure.{ x | x <- SHARD_FM, x == O or x == V} | ok

channel input, output: T1
PROCESS = input?x -> (output!x -> PROCESS [] output!failure.O -> PROCESS [] output!failure.V -> PROCESS)

channel input1, input2, input3: T1
channel result: T1
VOTER = input1?x1 -> input2?x2 -> input3?x3 ->(if f(x1)==0 and f(x2)==0 and f(x3)==0 -- no faulty inputs
                                                   then result!ok -> VOTER

                                          -- only one faulty inputs exist
                                          else if (f(x1)==5 and f(x2) == 0 and f(x3) == 0)
                                               or (f(x2)==5 and f(x1) == 0 and f(x3) == 0)
                                               or (f(x3)==5 and f(x1) == 0 and f(x2) == 0)
                                                   then result!ok -> VOTER

                                          -- two or more faulty inputs arrive
                                          else if (f(x1) ==5 and f(x2) == 5) or
                                                  (f(x1) ==5 and f(x3) == 5) or
                                                  (f(x2) ==5 and f(x3) == 5)
                                                   then result!failure.V -> VOTER []
                                                        result!failure.O -> VOTER

                                          -- any omission of inputs or failures of hardware
                                          else result!failure.O -> VOTER)
```

Figure 3. Definition of failure behaviours in the TMR example

3.4 Compositional Failure Modelling

The definition of CSP processes is compositional: two processes can be combined to form a larger process in a white-box or black-box view using parallel composition operators. The fundamental rule of process composition is again synchronisation: processes to be composed require synchronisation on the events they allow. Within the framework of failure modelling, each synchronisation point (i.e., interaction point of processes) represents the means to failure propagation across a predetermined boundary between components, whereas unsynchronised failure events such as internally generated events are permitted to occur freely only within the component boundary. When putting all elementary failure processes into the entire system, we can see directly how the occurrence of an external or internal failure propagates to the system boundary and affects safe system operation conditions.

Given the same TMR example illustrated in Figures 2 and 3, there is only one composite process – the TMR_CCVIEW process representing failure behaviour of the C&C view of the TMR system, which composes the four elementary processes P1, P2, P3 and V1. The three redundant processes P1, P2 and P3 interact with the V1 process through their outputs, whilst there is no interaction between P1, P2 and P3 processes. Hence, the resulting definition of the TMR_CCVIEW can be shown as follows:

TMR_CCVIEW = (P1 ||| P2 ||| P3) [|{|out1, out2, out3|}|] V1

Composition can also be applied to composite processes such as failure model views and subsystems. In general, the composition of views is similar to the one of components and requires the provision of synchronisation points. In many cases, there are no suitable synchronisation points between views, and thus the mapping between them needs to be defined before the composition operation. For example, conceptual view and hardware view cannot be composed directly without the definition of their mapping (i.e., an allocation view).

Once a failure process is defined, it can be stored and re-used in the future using the CSP renaming operator.

3.5 Causality

The basic tenet of CSP is communication and there are two fundamental laws of causality within its communication framework:

- *Temporal ordering* in the sense that a -> b -> P implies that if event *b* happens then event *a* must have happened.
- *Handshaking synchronisation* in that events only happen when both parties agree

Obviously, temporal ordering is a necessary condition of causality (i.e., causes must happen before effects), and the synchronisation semantics restricts further the event ordering in terms of interactions of processes. The resulting model of CSP causality is the trace model, in which the set of all possible sequences of events (i.e., traces) of a CSP process can be derived. The trace model is only a part of causal model in the sense that there is no way of determining whether event *a* causes event *b* by the observation that *b* always follows by *a*. Although we often assume causal relations implicitly when modelling each failure flow by the CSP sequential notation, causality cannot be maintained upon the composition of various failure flows

Adding causal semantics into CSP language involves changes to CSP notation. Instead of doing this, we propose to conclude

causal relationships from analysis of the existing trace model. Although event *b* following by event *a* cannot say *a* causes *b* to happen, we can find out the potential underlying causality by changing the states of event sequences to see its results. This idea is borrowed from philosophy domain (i.e., there is a causal connection from A to B if and only if we can change B by changing A [1]). For example, consider the following traces:

<input.failure.O, a.ok, b.ok, output.ok>,
<input.failure.O, a.ok, b.fail, output.failure.V>
<input.failure.O, a,fail, b.ok, output.failure.V>
<input.failure.O, a.fail, b.fail, output.failure.V>

By isolating the initiating event *input.failure.O*, a causal relation between undesired system event *output.failure.V* and failure events *a.fail* and/or *b.fail* could be established. Note that *a.ok* or *b.ok* can be thought as the non-occurrence of *a.fail* or *b.fail*. Since the CSP external choice denotes alternatives of behaviours and the sequential notation represents the enabling condition of the event occurrences, we here can treat the external choice notation as logical disjunction and the sequential notation as conjunction. As a result, we have the following expression results:

$$\begin{aligned}\text{occur(output.failure.V)} &= (\text{occur(a.ok)} \wedge \text{occur(b.fail)}) \vee \\ &\quad (\text{occur(a.fail)} \wedge \text{occur(b.ok)}) \vee \\ &\quad (\text{occur(a.fail)} \wedge \text{occur(b.fail)}) \\ &= \text{occur(a.fail)} \wedge \text{occur(b.fail)}\end{aligned}$$

We can now conclude that either the failure of *a* device or the one of *b* device can lead to the undesired event *output.failureV* under the initiating condition (arrival of omission failure at input). It can be envisioned that the dependency between events can also be discovered in a similar way.

3.6 Use of CSP Tools

One of the main benefits using CSP for failure modelling is the availability of tool support. Our experiments have applied two common CSP tools (ProBE and FDR2) in failure modelling. The animator ProBE is used to validate intended failure behaviour of a component by giving the user the chance to explore it in an interactive mode, whilst the model checker FDR2 verifies the consistency of failure models by deadlock checking. There are also two important applications of FDR2 in our work on failure modelling. The first one is refinement checking to ensure consistency between views using the CSP hiding operator. For example, the allocation failure view refines the C&C failure view by adding the details of hardware failure events.

```
assert TMR_CCVIEW [T= TMR_ALLOCVIEW \ ICpu
```

Another application is to find failure scenarios of interest by generating counterexamples. In the latter case, we specify desired safety properties of the failure model that exclude the events of interest (i.e., undesired system-level failures) and check it (traces refinement) against the failure process. The property should not hold and counterexamples can be generated by FDR2. The batch interface of FDR2 provides direct control on the generation of counterexamples and thus automated analysis on the generated failure scenarios is possible. The following is an example property to check the system-level value failures in the TMR system within the C&C view:

```
ISafeSys = diff(Events, {output.failure.V})
-- anything but value failures of output allowed
SAFESPEC = [] x : ISafeSys @ x -> SAFESPEC
assert SAFESPEC [T= TMR_CCVIEW
```

4. INITIAL RESULTS

To evaluate the effectiveness of the proposed method, we have applied it to a safety-related software system, in which an initial software architecture has been developed. The architectural notation adopted was UML-RT, and only two architectural views (C&C and allocation views) were considered. Our findings reveal that a hardware view must be provided in order to understand the relationship between hardware failure flows. The hardware view can be derived from either the allocation view or hardware architecture description. The choice of architectural descriptions or representations is not important to our method, since users are free to define their own architectural transformation rules. Also, the architecture doesn't need to be complete and can be analysed selectively.

One difficulty we encountered is lack of development tool support for CSP modelling (and specifically failure modelling). Definition of failure behaviour can be complicated, large and potentially error-prone. Using FDR2 to check errors in definition of complex or large processes can be time-consuming. This has been acknowledged in security protocol analysis using CSP [18]. Another problem is the inherent communication nature of CSP. The communicating problems such as deadlocks and livelocks are themselves safety problems from the viewpoint of system functionality, but are no longer meaningful within our failure-modelling framework that aims to identify and reason about the event combinations that would lead to system failures. A deadlocked process indicates only a CSP modelling problem rather than the problems of failure mechanisms of the system. Finally, discovering causality based on the CSP trace model will require further tool development.

5. RELATED WORK

Besides formal specification of system functionality, there has been another important application of CSP in the safety community: namely, specifying fault tolerance [21] to prove that a fault-tolerant system will behave as intended even in the presence of faults. This view is limited to functional safety and differs fundamentally from our goal to identify possible failure combinations leading to undesirable system events.

Our work has been strongly influenced by York's existing work on Failure Propagation and Transformation Notation (FPTN) [10]. In FPTN, safety analysis is structured in response to system structure (i.e., modules) and failure behaviour of each module is described in terms of failure flows, which can be classified into four forms: failure propagation, failure transformation, failure handling and failure generation. Our work can be seen as an evolution of FPTN.

The ESACS project [7] proposed two alternatives of failure modelling: system model prototyping for safety – using predefined components enriched with failure modes to perform fast prototypes, or failure mode injection – extending the system model by injecting failure modes. Both of them inspire our work on construction of failure models. A number of state-based formalisms such as AltaRica [5] have been proposed for system modelling for safety. Our position is that, in many cases, we

cannot assume knowledge of internal details of components such as Commercial-Off-The-Shelf (COTS) components. For this reason, we prefer an approach based on observable behaviours, rather than internal states and state transitions.

Causality has been long recognised as the core concept of safety analysis in the safety community. A number of causal analysis techniques have been proposed for accident modelling and analysis, such as STAMP [16] and Why-Because Analysis [14]. Many of these techniques are based on the counter-factual arguments in the form of "if X did not occur then the accident would have been avoided" [13], which is very closely related to the technique for discovering causality in our approach.

6. CONCLUSIONS AND ONGOING WORK

In this paper, we have presented the application of CSP to failure modelling in the context of software architecture design. The flexible modelling capabilities of CSP and powerful tool support have driven our work to extend system modelling in CSP to include failure behaviour. These extensions provide an effective means for both software architects and safety engineers to evaluate safety-related architectural decisions and identify credible failure mechanisms of the system.

We are currently examining the possibility of generating CSP failure models from architectural models annotated with necessary failure information such as failure modes and failure propagation rules. Our ongoing work also includes extensions to model probabilistic aspects of failure behaviours in order to facilitate both qualitative and quantitative analyses, and developing tools for generating safety analysis results such as FTA, Failure Modes and Effects Analysis (FMEA) and Markov Analysis (MA) from the failure models.

7. ACKNOWLEDGMENTS

The authors would like to thank Airbus UK for supporting this research, and others working in the same area in the HISE group.

8. REFERENCES

[1] Causation. in Sosa, E. and Tooley, M. eds., Oxford University Press, Oxford ; New York, 1993.

[2] *Failures-Divergence Refinement. FDR2 User Manual*. Formal Systems (Europe) Ltd, 2003.

[3] *Process Behaviour Explorer. ProBE User Manual*. Formal Systems (Europe) Ltd, 2003.

[4] Allen, R.J. *A Formal Approach to Software Architecture, PhD thesis* thesis, Software Engineering Institute, Carnegie Mellon University, 1997.

[5] Arnold, A., Griffault, A., Point, G. and Rauzy, A. AltaRica formalism for describing concurrent systems. *Fundamenta Informaticae*, *40*, 2000, 109-124.

[6] Bass, L., Clements, P. and Kazman, R. *Software Architecture in Practice, 2nd Edition*. Addison Wesley, Reading, MA, USA, 2003.

[7] Bozzano, M., Villafiorita, A., Åkerlund, O., Bieber, P., Bougnol, C., Böde, E., Bretschneider, M., Cavallo, A., Castel, C., Cifaldi, M., Cimatti, A., Griffault, A., Kehren, C., Lawrence, B., Lüdtke, A., Metge, S., Papadopoulos, C., Passarello, R., Peikenkamp, T., Persson, P., Seguin, C., Trotta, L., Valacca, L. and Zacco, G., ESACS: an integrated methodology for design and safety analysis of complex systems. in *Proceedings of European Safety and Reliability Conference (ESREL 2003)*, Balkema Publisher.

[8] Easterbrook, S., Lutz, R., Covington, R., Kelly, J., Ampo, Y. and Hamilton, D. Experiences Using Lightweight Formal Methods for Requirements Modeling. *IEEE Transactions on Software Engineering*, *24* (1), 1998, 4-14.

[9] Feiler, P.H., Lewis, B. and Vestal, S., The SAE Avionics Architecture Description Language (AADL) Standard: A Basis for Model-Based Architecture-Driven Embedded Systems Engineering. In *RTAS 2003 Workshop on Model-Driven Embedded Systems*, (2003).

[10] Fenelon, P. and McDermid, J. An integrated toolset for software safety analysis. *Journal of Systems and Software*, *21* (3), 1993, 279-290.

[11] Hoare, C.A.R. *Communicating sequential processes*. Prentice Hall International, Englewood Cliffs, N.J., 1985.

[12] Hofmeister, C., Nord, R.L. and Soni., D., Describing software architecture with UML. in *Proceedings of the TC2 1st Working IFIP Conference on Software Architecture (WICSA1)*, (San Antonio, TX, USA, 1999), 145-159.

[13] Johnson, C., The Application of Causal Analysis Techniques for Computer-Related Mishaps. in *Proceedings of the 22nd International Conference on Computer Safety, Reliability and Security (SAFECOMP'03)*, (Edinburgh, UK, 2003), Springer Verlag, 368-381.

[14] Ladkin, P. and Loer, K. *Why-Because Analysis: Formal Reasoning About Incidents*. Technischen Fakultat der Universitat Bielefeld, Germany, Bielefeld, Technical Report, Document RVS-Bk-98-01, 1998.

[15] Leveson, N.G. *Safeware: System Safety and Computers*. Addison-Wesley, 1995.

[16] Leveson, N., A Systems Model of Accidents. in *Proceedings of the 20th International Conference of the System Safety Society*, (Unionville, U.S.A., 2002), International Systems Safety Society, 476-486.

[17] Lions, J.L. *ARIANE 5: Flight 501 failure*, Paris, Ariane 5 Inquiry Board Report, 1996.

[18] Lowe, G. Casper: A compiler for the analysis of security protocols. *Journal of Computer Security*, *6*, 1998, 53-84.

[19] McDermid, J.A. and Pumfrey, D.J., A Development of Hazard Analysis to aid Software Design. in *Proceedings of the Ninth Annual Conference on Computer Assurance (COMPASS '94)*, (Gaithersburg, 1994), IEEE, 17-25.

[20] Perrow, C. *Normal Accidents: Living with High Risk Technologies,*. Basic Books, New York, 1984.

[21] Roscoe, A.W. *The theory and practice of concurrency*. Prentice Hall, London, 1998.

[22] Selic, B. *UML-RT: A profile for modeling complex real-time architectures*. ObjecTime Limited, 1999.

[23] Wu, W. and Kelly, T., Safety Tactics for Software Architecture Design. In *Proceedings of the 28th Annual International Computer Software and Applications Conference (COMPSAC 2004)*, (Hong Kong, 2004), IEEE Computer Society.

[24] Voas, J.M. *Software fault injection: inoculating programs against errors*. Wiley, New York, 1998.