Safety Interfaces for Component-Based Systems

Jonas Elmqvist[1] and Simin Nadjm-Tehrani[1] and Marius Minea[2]

[1] Department of Computer and Information Science, Linköping University
jonel,simin@ida.liu.se
[2] "Politehnica" University of Timişoara and Institute e-Austria Timişoara
marius@cs.utt.ro

Abstract. This paper addresses the problems appearing in component-based development of safety-critical systems. We aim at efficient reasoning about safety at system level while adding or replacing components. For safety-related reasoning it does not suffice to consider functioning components in their "intended" environments but also the behaviour of components in presence of single or multiple faults.
Our contribution is a formal component model that includes the notion of a safety interface. It describes how the component behaves with respect to violation of a given system-level property in presence of faults in its environment. We also present an algorithm for deriving safety interfaces given a particular safety property and fault modes for the component. Moreover, we present compositional proof rules that can be applied to reason about the fault tolerance of the composed system by analyzing the safety interfaces of the components. Finally, we evaluate the above technique in a real aerospace application.

1 Introduction

Component-based software development [30, 9] uses various models and methods to capture different attributes of a system, or emphasise phases of the development cycle [4, 28, 8, 27, 10]. This paper addresses efficient assurance of dependability in a system built from components and with several upgrades in its life cycle, an aspect not widely studied so far in the components literature [11].

Modifying a component or replacing it with another is an especially costly process for safety-critical systems, as much of the analysis and review of the safety arguments at the certification stage has to be repeated for every significant change to the system. We believe that tool support in this sector needs to make component changes cost-efficient by addressing safety-specific issues, e.g. resilience of the system with respect to single and multiple faults as new components are plugged in. The model we propose covers digital components, with a built-in declaration of their behaviour under faults in assumed environments. This component model captures the logic of the design at a high abstraction level, and could be applied to software or (reconfigurable) hardware designs.

Traditional risk assessment techniques such as Fault-tree analysis (FTA) and Failure modes and effects analysis (FMEA) [16] deal with the effect of independent faults. Although assessing fault tolerance at system level is an important part of safety analysis, rigorous methods are only in their infancy when

it comes to systems with significant digital components [15, 13]. Our goal is to provide a formal means to support the system integrator. When acquiring a new component for inclusion into a system, the integrator is informed whether the component can potentially threaten the system-level safety (in the same spirit as FMEA). The integrator is also supported in analysis of fault tolerance at system level, the result of which will indicate all single or multiple component-level faults that will necessarily lead to violation of safety (in the same vein as FTA). Unlike functional correctness analysis, here the goal is to focus on risks associated with external faults, not to eliminate design faults.

The contributions of this paper are as follows. We present a component model that includes *safety interfaces*. These describe how a component behaves with respect to a given system-level safety property in presence of (a defined set of) faults in its environment. We show how to perform a system-level safety analysis by using the safety interfaces of components. This goal is supported in two ways. First, we provide an algorithm that derives the safety interface of a component given a particular safety property and set of fault modes. The interface includes the single and multiple faults that this component is resilient to, as well as environment restrictions that can contain the faults. This analysis is intended to be performed by the developer of the component. Second, we support the system integrator to reason compositionally about safety in presence of single and multiple faults at system level by referring to the safety interfaces. Once the relevant fault-failure chains are rigorously identified, they can be handled using standard assessment routines, fault forecasting and containment techniques [20].

1.1 Related work

To our knowledge, there is no previous formal work on safety interfaces for components.

Current engineering practice includes two parallel activities for safety-related studies (hazard analysis, FTA and FMEA) and functional design and analysis. Recent research efforts have tried to combine these separate tracks by augmenting the system design model with specific fault modes. Åkerlund et al. [2] have to our knowledge the first attempt to integrate the separate activities of design and safety analysis and support them by common formal models. Hammarberg and Nadjm-Tehrani extend this work to models at a higher level of abstraction and characterise patterns for safety analysis of digital modules [15]. That work, however, does not build on a notion of encapsulation as in components. It verifies the entire composed system in Esterel using a SAT model checker and iteratively analyses all fault modes at system level. The ESACS project [7] applies a similar approach to Statechart models using a BDD-based verification engine.

Papadopolous et. al. [24] extend a functional model of a design with Interface-Focused FMEA. The approach follows a tabular (spread sheet) editor layout. The formalised syntax of the failure classes allows an automatic synthesis of a fault tree and incorporation of knowledge about the architectural support for mitigation and containment of faults. However, it suffers from combinatorial explosion in large fault trees and lacks formal verification support.

Rauzy models the system in a version of mode automata and the failure of each component by an event that takes the system into a failure mode [26]. The formal model is compiled into Boolean equations and partial order techniques are suggested for reducing the combinatorial explosion. However, it has not been applied to component-based development or compositional reasoning.

Strunk and Knight define the system and its reconfiguration elements explicitly using RTL (temporal logic) notation and provide guidelines for reconfiguration assurance. Reconfiguration is mainly used here when the system is adapting to lower service levels that may in particular be due to failure scenarios [29].

Jürjens defines an extension of the UML syntax in which stereotypes, tags, and values can be used to capture failure modes of components in a system (corruption, delay, loss) [19]. The merit of the model is to narrow the gap between a system realised as a set of functions and a system realised as a set of components.

Li et al. [21] define feature-oriented interfaces for modules that encapsulate crosscutting system properties. The focus of this work is feature interaction including features that introduce a new vocabulary.

A recent approach for formal treatment of crosscutting concerns in reconfigurable components is given by Tesanovic et al. [31] where extended timed automata are used to capture models of components with an interface for characterising the essential traces for supporting a given *timing property*.

Assume-guarantee-style compositional reasoning has a long history originating with the work by Misra and Chandy [23] and Jones [18] in the context of concurrent systems. It has been applied to deductive reasoning about specifications [1] as well as model checking for various automata formalisms. Here, the notion of refinement is usually trace inclusion, but can also be simulation [17]. Our rules are derived from those of Alur and Henzinger for reactive modules [3].

2 Components and fault models

A component is an independent entity that communicates through well-defined interfaces. In most component models, the interfaces are only functional, defining input and output ports at a syntactic level. For efficient safety analysis at system level, these simple interfaces are insufficient. More behaviour information must be provided to make interfaces usable for analysis of failures in presence of faults.

We propose a formal component model with two elements: its functional behaviour and a *safety interface*, which describes the behaviour in presence of faults in the environment. This safety interface can then be used to perform safety analysis at system level, such as analysis for fault tolerance. We next present the basic definitions, the fault modes and the employed formalism.

2.1 Modules and basic definitions

Our general formalism for modules is based on the notion of *reactive modules* [3], of which we give only a brief overview. We present a special class of reactive

modules with synchronous composition, finite variable domains and non-blocking transitions that we call *synchronous modules* (by default, simply modules).

A module is defined by its *input*, *output* and *private variables* and the rules for updating and initializing them. Variables are updated in a sequence of rounds, each once per round. To model synchrony, each round is divided into subrounds, and the system and the environment take turns in executing and updating variables. Events, such as a *tick*, can be modelled by toggling boolean variables.

Definition 1 (Module). *A synchronous module M is a tuple (V, Q_0, δ) where*

- *$V = (V_i, V_o, V_p)$ is a set of typed variables, partitioned into sets of input variables V_i, output variables V_o and private variables V_p. The controlled variables are $V_{ctrl} = V_o \cup V_p$ and the observable variables are $V_{obs} = V_i \cup V_o$;*
- *A* state *over V is a function mapping variables to their values. The set of controlled states over V_{ctrl} is denoted Q_{ctrl} and the set of input states over V_i as Q_i. The set of states for M is $Q_M = Q_{ctrl} \times Q_i$;*
- *$Q_0 \subseteq Q_{ctrl}$ is the set of initial states;*
- *$\delta \subseteq Q_{ctrl} \times Q_i \times Q_{ctrl}$ is the* transition relation.

The successor of a state is obtained at each round by updating the controlled variables of the module. The execution of a module produces a state sequence $\bar{q} = q_0 \dots q_n$. A trace $\bar{\sigma}$ is the corresponding sequence of observations on $\bar{q}$, with $\bar{\sigma} = q_0[V_{obs}] \dots q_n[V_{obs}]$, where $q[V']$ is the projection of q onto a set of variables $V' \in V$. The trace language of M, denoted $\mathcal{L}_M$, is the set of traces of M.

A property φ on a set of variables V is defined as a set of traces on V. A module M satisfies a property φ, written $M \models \varphi$, if all traces of M belong to φ. This work focuses on *safety properties* [22, 14] as opposed to liveness properties.

Composing two modules into a single module creates a new module whose behaviour captures the interaction between the component modules.

Definition 2 (Parallel composition). *Let $M = (V^M, Q_0^M, \delta^M)$ and $N = (V^N, Q_0^N, \delta^N)$ be two modules with $V_{ctrl}^M \cap V_{ctrl}^N = \emptyset$. The parallel composition of M and N, denoted by $M \parallel N$, is defined as*

- *$V_p = V_p^M \cup V_p^N$*
- *$V_o = V_o^M \cup V_o^N$*
- *$V_i = (V_i^M \cup V_i^N) \setminus V_o$*
- *$Q_0 = Q_0^M \times Q_0^N$*
- *$\delta \subseteq Q_{ctrl} \times Q_i \times Q_{ctrl}$ where $(q, i, q') \in \delta$ if $(q[V_{ctrl}^M], (i \cup q)[V_i^M], q'[V_{ctrl}^M]) \in \delta^M$ and $(q[V_{ctrl}^N], (i \cup q)[V_i^N], q'[V_{ctrl}^N]) \in \delta^N$.*

We extend Definition 2 to a pair of modules with shared outputs, provided the resulting transition relation δ is *nonblocking*, i.e., has a next state for any combination of current state and inputs. In this case, we call the two modules *compatible* and distinguish *nonblocking composition* by denoting it $\widehat{\parallel}$.

We relate modules via trace semantics: a module M refines a module N if N has more behaviours than M, i.e., all possible traces of M are also traces of N.

Definition 3 (Refinement). *Let $M = (V^M, Q_0^M, \delta^M)$ and $N = (V^N, Q_0^N, \delta^N)$ be two synchronous modules. M refines N, written $M \leq N$, if (1) $V_o^N \subseteq V_o^M$, (2) $V_{obs}^N \subseteq V_{obs}^M$ and (3) $\{\bar{\sigma}[V_{obs}^N] : \bar{\sigma} \in \mathcal{L}_M\} \subseteq \mathcal{L}_N$.*

2.2 Fault mode models

To analyse the behaviour of a component in presence of faults in its environment it is important to identify all possible ways that the environment can fail. Low-level fault modes are generally application and platform dependent, however faults can be classified into high-level categories. Bondavalli and Simoncini classify faults into *omission faults*, *value faults* and *timing faults* [6]. We adopt a classification in which faults fall into the following categories [12, 25, 24]:

Omission failure i.e., absence of a signal when the signal was expected.
Commission failure i.e., unexpected emission of a signal.
Value failure i.e., failure in the value domain such as signal out of range or a signal stuck at some value etc.
Timing failure i.e., failure in the time domain such as late or early delivery.

Timing properties have been addressed in other work, for example interfaces to capture timing properties in the absence of faults are given by Tesanovic et al. in timed automata [31] and could be further extended to cover resilience to timing failures. In this work we focus on untimed models and value failures.

We model faults in the environment as delivery of faulty input to the component and call each such faulty input a *fault mode* for the component. A value failure is modelled by modifying the input signals that in turn might affect private variables. A commission failure is modelled by unforced emission of signals to the component. The input fault of one component thereby captures the output fault of a component connecting to it, with the exception of "edge" components that need to be treated separately, e.g. in accordance to earlier methods [15].

Definition 4 (Input Fault Mode). *An input fault mode F_j of a module M is a module with an input variable $v_j^f \notin V^M$, an output variable $v_j \in V_i^M$, both of the same type D, and an unconstrained transition relation $\delta = D \times D \times D$.*

A fault mode F_j on the input v_j from environment E to module M can be viewed as replacing the original output v_j of E with the input v_j^f of F_j, which produces the faulty output v_j to M. We model this formally as a composition of F_j and E, which has the same variables as E and can then be composed with M. Free inputs to M are viewed as unconstrained outputs of E.

Definition 5 (Composition with Fault). *Let E be a module with $v_j \in V_o^E$ and F_j a fault mode with output v_j and input v_j^f. Denote $F_j \circ E = F_j \parallel E[v_j/v_j^f]$ where $E[v_j/v_j^f]$ is the module E with the variable substitution v_j^f for v_j.*

Our fault modes are unrestricted and can affect their output in an arbitrary way. Other types of fault modes can be modelled by appropriate logic in their transition relation. We can naturally extend this definition to multiple faults.

2.3 Components and safety interfaces

Given a module, we wish to characterize its fault tolerance in an environment that represents the remainder of the system together with any external constraints. Whereas a module represents an implementation, we wish to define an interface that provides all information about the component that the system integrator needs. Traditionally, these interfaces do not contain information about safety of the component. In this paper we propose a safety interface that captures the behaviour of the component in presence of faults in the environment.

Definition 6 (Safety Interface). *Given a module M, a system-level safety property φ, and a set of fault modes F for M, a safety interface SI^{φ} for M is a tuple $\langle E^{\varphi}, \mathsf{single}, \mathsf{double}\rangle$ where*

- *E^{φ} is an environment in which $M \parallel E^{\varphi} \models \varphi$.*
- *$\mathsf{single} = \langle F^s, E^s\rangle$ where $F^s \subseteq \mathcal{P}(F)$ is the* single fault resilience set *and E^s is a module composable with M, such that $\forall F_k \in F^s, M \parallel (F_k \circ E^s) \models \varphi$*
- *$\mathsf{double} = \{\langle F_1^d, E_1^d\rangle, \ldots, \langle F_n^d, E_n^d\rangle\}$ with $F_k^d = \langle F_k^1, F_k^2\rangle$, $F_k^1, F_k^2 \in F$, $F_k^1 \neq F_k^2$ such that $M \parallel ((F_k^1 \parallel F_k^2) \circ E_k^d) \models \varphi$*

The safety interface makes explicit which single and double faults the component can tolerate, and the corresponding environments capture the assumptions that M requires for resilience to these faults. For single faults, we specify *one* environment assumption E^s under which the component is resilient to *any* fault from a given set of interest. For double faults, we are more fine-grained and specify for each fault pair of interest an environment in which the module is resilient to their joint occurrence. Multiple faults could be handled similarly. The safety interface need not cover all possible faults (and in fact could be empty): the provider of a component only specifies what is explicitly known about it.

Definition 7 (Component). *Let φ be a system-level safety property, M a module and SI^{φ} a safety interface for M. A component C is the tuple $\langle M, SI^{\varphi}\rangle$.*

We wish to deliver a component with precomputed information about the set of tolerated fault modes. To check safe use of the component one verifies that the actual environment satisfies the component assumptions which guarantee safety under faults.

3 Deriving safety interfaces

In this section we provide guidelines for how a component developer creates the safety interface. The developer needs to characterise environments in which a module functions correctly in the presence of a given set of faults. We first derive such an environment (in fact, the most general one) in the ideal case without faults. Next, we use the obtained environment abstraction to determine more restrictive environments under which the module is resilient, first to a chosen set of single faults and then for the occurrence of fault pairs.

3.1 Generating a constraining environment

If M is a module such that $M \not\models \neg\varphi$, the weakest (least restrictive) environment E_w^φ in order to satisfy φ can be generated as shown in Figure 1. The algorithm uses a model checker to check whether the module M in parallel with an environment E satisfies the safety property φ; i.e. $M \parallel E \models \varphi$.

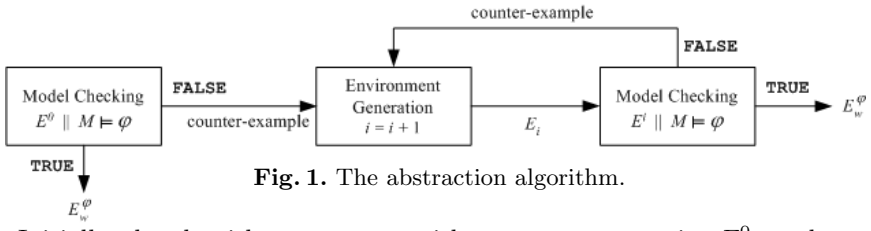

Fig. 1. The abstraction algorithm.

Initially, the algorithm starts out with an empty constraint E^0 on the environment and at each iteration i, the algorithm strengthens the constraints E^i by analysing the counter-example generated by the model checker and removing the forbidden states. This corresponds to removing behaviour from (or strengthening) the environment. In the next iteration, the environment E^{i+1} should at least not exhibit the behaviour reflected by the counter-example at iteration i. The algorithm stops at a fixpoint when $E^{i+1} = E^i = E_w^\varphi$.

Proposition 1. *The environment E_w^φ generated by the algorithm is the* least restrictive environment *in which M satisifies the property φ. That is, for any environment E, $M \parallel E \models \varphi$ iff $E \leq E_w^\varphi$.*

The proof can be done by adapting the reasoning by Halbwachs et. al. [14], that synthesise a necessary and sufficient environment for an I/O machine M.

3.2 Identification of fault behaviours

Let M be a module that satisfies a safety property φ when placed in an environment E, assuming the ideal case without faults: $M \parallel E \models \varphi$. Let F_j be a fault mode on variable v_j which is an input to M from E. Denote by $E' = \forall v_j\, E$ the module with $V^{E'} = V^E \setminus v_j$, $Q_0^{E'} = \forall v_j\, Q_0^E$ and $\delta^{E'} = \forall v_j\, \forall v_j'\, \delta^E$.

Proposition 2. *If $M \parallel E \models \varphi$ and $\forall v_i\, E$ exists, then $M \parallel (F_i \circ \forall v_i\ E) \models \varphi$, i.e., M is resilient to fault F_i in the environment $\forall v_i\, E$.*

By definition of $\forall v_i\, E$, any state can be extended to a state of E with an arbitrary value of v_i. Thus, $F_i \circ \forall v_i\, E \leq E$ and the result follows by composing with M. In particular, if E_w^φ is the least restrictive environment for M and φ, then $\forall v_i\, E_w^\varphi$ is the least restrictive environment in which module M is resilient to fault F_i.

This result gives an environment in which a module is resilient to a single fault. For the safety interface, we need an environment E^s which makes the module resilient to *any one* fault from the *single fault resilient set*. The desired environment must be at least as restrictive as the environment required for resilience of each of the individual fault modes. This is ensured by their parallel composition.

Proposition 3. *If* $M \parallel (F_i \circ E_i) \models \varphi$ *and* $M \parallel (F_j \circ E_j) \models \varphi$*, and* E_i*,* E_j *are compatible, then* $M \parallel (E_i \mathbin{\widehat{\parallel}} E_j)$ *is resilient to any fault* F_i *or* F_j *individually.*

This follows since any trace of $E_i \mathbin{\widehat{\parallel}} E_j$ under fault F_i or F_j is either a trace of $F_i \circ E_i$ or of $F_j \circ E_j$. In particular, we can take $E_i = \forall v_i\, E_w^{\varphi}$ with E_w^{φ} the least restrictive environment as determined in the previous section. Successive application to each fault in the selected set yields $E^s = \forall v_i\, E_w^{\varphi} \mathbin{\widehat{\parallel}} \ldots \mathbin{\widehat{\parallel}} \forall v_n\, E_w^{\varphi}$ as the desired environment for the single element of the safety interface.

For resilience to double faults F_i and F_j, the environment must be restricted, analogously to Proposition 2, to behaviours allowed for *all* values of v_i and v_j:

Proposition 4. *If* $M \parallel E \models \varphi$*, and* F_i, F_j *are faults such that* $\forall v_i \forall v_j\, E$ *exists, then* $M \parallel ((F_i \parallel F_j) \circ \forall v_i \forall v_j\, E) \models \varphi$ *. That is,* M *is resilient to simultaneous faults* F_i *and* F_j *in the environment* $\forall v_i \forall v_j\, E$*.*

Thus, if $\forall v_i \forall v_j\, E$ is nonempty, the pair $\langle \langle F_i, F_j \rangle, \forall v_i \forall v_j\, E \rangle$ can be included in the double fault resilience portion of the safety interface. Moreover, if E_w^{φ} is the least restrictive environment for M and φ, then $\forall v_i \forall v_j\, E_w^{\varphi}$ is the least restrictive environment in which M is simultaneously resilient to F_i and F_j.

Example: Suppose module M guarantees the safety property φ if the environment E ensures that of the two boolean inputs v_1 and v_2, at least one is set to 1: $v_1 \vee v_2 = 1$. Then, the faulty environments become $E_1 = F_1 \circ E \equiv \forall v_1\, E \equiv \forall v_1 \,.\, v_1 \vee v_2 = 1 \equiv v_2 = 1$ and $E_2 = F_2 \circ E \equiv v_1 = 1$. The environment which is resilient to either fault is $E_1 \mathbin{\widehat{\parallel}} E_2 \equiv v_1 = 1 \wedge v_2 = 1$. There is no environment under which the module is resilient to a double fault.

4 Component-based analysis of fault tolerance

We next describe the methodology of applying the above component model in system safety analysis. Unlike component models that capture functional contracts as interfaces, and then apply assume-guarantee reasoning for ensuring that the system behaves functionally correct when built from given components, our model does not aim to prove the satisfaction of a property. Rather, the purpose of our analysis is to focus on sensitive faults. In other words, both resilience and non-resilience information are interesting in the follow-up decisions. If the system safety is indeed threatened by a single fault, then the systems engineer may or may not be required to remove the risk of that fault by additional actions. This typically implies further qualitative analysis of the risk for the fault and its consequence, and is outside the scope of this paper. Combinations of multiple faults typically bear a lower risk probability but as important to quantify and analyse. If the safety of the system is not sensitive to a fault (pair), then the engineer can confidently concentrate on other combinations of potential faults that are a threat. In general, it is likely that none of these faults appear in actual operation, and the whole study is only hypothetical in order to provide arguments in preparing the safety case for certification purposes.

With this introduction, we will now proceed to explain the steps needed to ascertain the sensitivity of the system to single (respectively multiple) faults.

4.1 General setup

Consider a system safety property φ and a component with safety interface SI^{φ}. As delivered by the component provider, SI^{φ} specifies an environment in which the component is safe, assuming no faults; another environment in which the component is resilient to a set of single faults, and a safe environment for each considered pair of simultaneous faults. Consider proving $M_1 \parallel M_2 \parallel ... \parallel M_n \models \varphi$ in the presence of a fault F_j in M_1. If a safety interface SI^{φ} of M_1 is known, with $\mathsf{single} = \langle F_1^s, E_1^s \rangle$, and $F_j \in F_1^s$, it suffices to show that $M_2 \parallel \ldots \parallel M_n \leq E_1^s$.

However, composing all modules is against the idea of modular verification. This can be overcome using circular assume-guarantee rules [3], for which we first derive an n-module version. The rule requires that every module in its environment (an abstraction of the other modules) refines each other environment. We can then infer that the system refines the composition of the environments without paying the price of an expensive overall composition, and without having to redo the entire analysis each time a component changes.

Lemma 1. *Let M_j and E_j, $1 \leq j \leq n$ be modules and environments such that the compositions $I = M_1 \parallel \ldots \parallel M_n$ and $E = E_1 \mathbin{\widehat{\parallel}} \ldots \mathbin{\widehat{\parallel}} E_n$ exist and $V_j^E \subseteq V_{obs}^I$. Then, if $\forall i \forall j \; M_j \parallel E_j \leq E_k$ we have $M_1 \parallel \ldots \parallel M_n \leq E_1 \mathbin{\widehat{\parallel}} \ldots \mathbin{\widehat{\parallel}} E_n$.*

In concise rule form: $$\frac{\forall j \forall k \; M_j \parallel E_j \leq E_k}{M_1 \parallel \ldots \parallel M_n \leq E_1 \mathbin{\widehat{\parallel}} \ldots \mathbin{\widehat{\parallel}} E_n}$$

The proof follows that of Proposition 5 in [3], showing inductively that every step of the implementation I can be extended to a step of the specification E. Requiring nonblocking composition guarantees soundness despite circularity.

To reason compositionally about safety, we add n premises stating that each module in its given environment satisfies the safety property: $\forall i \; M_i \parallel E_i \models \varphi$. Together with the premises above, we can then prove safety for the composition:

Proposition 5. *If M_j and E_j, $1 \leq j \leq n$ satisfy the conditions of Lemma 1 and in addition $M_j \parallel E_j \models \varphi$ for $1 \leq j \leq n$ then we have $M_1 \parallel M_2 \parallel \ldots \parallel M_n \models \varphi$.*

In concise form: $$\frac{\forall j \; M_j \parallel E_j \models \varphi \qquad \forall j \forall k \; M_j \parallel E_j \leq E_k}{M_1 \parallel M_2 \parallel \ldots \parallel M_n \models \varphi}$$

Proof. Composing $M_j \parallel E_j \models \varphi$ for $j = 1...n$ we get $I \mathbin{\widehat{\parallel}} E \models \varphi$. By Lemma 1, $M_1 \parallel \ldots \parallel M_n \leq E_1 \mathbin{\widehat{\parallel}} \ldots \mathbin{\widehat{\parallel}} E_n$, or $I \leq E$. Thus $I \mathbin{\widehat{\parallel}} I \models \varphi$ or $I \models \varphi$.

This rule provides a generic assume-guarantee framework, independent of faults. We need to discharge n^2 premises to prove the global property φ, but each of those involves only one module, and at most two environment abstractions, assumed to be much smaller than the global composition. To use the rule, we need to find appropriate environments E_i, and to apply it to system safety, the environments must make the premises hold even with the analyzed fault(s) occurring. We derive these environments from the component safety interfaces.

4.2 Single faults

We assume that single faults affect only one component. Using the environments E^{φ} and E^{s}, we check safety compositionally showing the premises of Prop. 5:
– a module in a faulty environment still provides an environment that guarantees the safety of each other module in absence of another fault
– a module in a non-faulty environment provides for each other module the environment of the SI which makes it resilient to single faults.
Thus, we need to show premises (a) $M_j \parallel F \circ E^s_j \leq E^{\varphi}_k$ and (b) $M_k \parallel E^{\varphi}_k \leq E^s_j$, ranging over modules M_j with potential faults F, and non-faulty modules M_k. If the interface provides the weakest environment E^{φ}_w, the fault-specific premises (a) can be jointly replaced by $M_j \parallel E^{\varphi}_j \leq E^{\varphi}_k$, with fewer obligations to discharge.

4.3 Multiple faults

Next we study whether two faults F_a and F_b appearing in *different* components can, together, violate system safety. In Proposition 5 we use different environments for each component, depending on how they are affected by faults:
– for a module M_i affected by both faults, we check whether the double part of the safety interface contains a tuple $\langle\langle F_a, F_b\rangle, E^{ab}_i\rangle$, and use environment E^{ab}_i.
– for a module M_j with only one fault F_a, we use environment $E^a = \forall v_a E^{\varphi}_j$.
– for a module M_k not affected by faults, we use the environment E^{φ}_k.

Here, we have more fault-specific premises, but since environments for double faults are more restrictive, some premises can subsume or be subsumed by those for single faults. Thus, $M_i \parallel E^{ab}_i \leq E^{\varphi}_k$ follows from $M_i \parallel E^a_i \leq E^{\varphi}_k$, and checking $M_k \parallel E^{\varphi}_k \leq E^{ab}_i$ subsumes checking for single faults F_a or F_b.

5 Application: JAS Leakage Detection Subsystem

As a proof of concept we have applied our method to the leakage detection subsystem of the hydraulic system of the JAS 39 Gripen multi-role aircraft, obtained from the Aerospace division of SAAB AB [15]. Both the original system model and our component-based version are described in Esterel [5], a synchronous language whose compiler ensures the nonblocking property upon composition.

5.1 Functionality and safety

The system's purpose is to detect and stop potential leakages in two hydraulic systems (HS1 and HS2) that provide certain moving parts of the aircraft with mechanical power. Leakages in the hydraulic system could in the worst case lead to such a low hydraulic pressure that the aircraft becomes uncontrollable. To avoid this, four shut-off valves protect some of the branching oil pipes to ensure that at least the other branches keep pressure and supply the moving parts with power if a leakage is detected. However, closing more than one shut-off valve at the same time could result in locking the flight control surfaces and the landing gear which could have disastrous effects. Thus, overall aircraft safety depends on the property φ: *no more than one valve should be closed at the same time.*

5.2 Architectural view

The electronic part of the leakage detection subsystem consists of three electronic components (H-ECU, PLD1 and PLD2), four valves and two sets of sensors (see Figure 2). The H-ECU continually reads the oil reservoir levels of the two hydraulic systems, determines if there is a leakage, and if so, initialises a shut-off sequence of the valves. To ensure that the overall property is satisfied, two programmable logic devices, PLD1 and PLD2, continually read the status of the valves and send signals to them as well. If the readings indicate that more than two valves will close, PLD1 and PLD2 will disallow further closing of any valves. Thus, PLD1 and PLD2 increase the fault tolerance of the shut-off subsystem implemented in the H-ECU.

Each valve is controlled by two electrical signals, one signal on the high side from the PLD2 and one on the low side from the H-ECU. Both of these signals need to be present in order for the valve to close. In this study, we only consider the three components H-ECU, PLD1 and PLD2. Thus, due to the functionality of the valves, the property φ can be replaced by φ': *no more than one valve should receive signals on both the high side and the low side at the same time.*

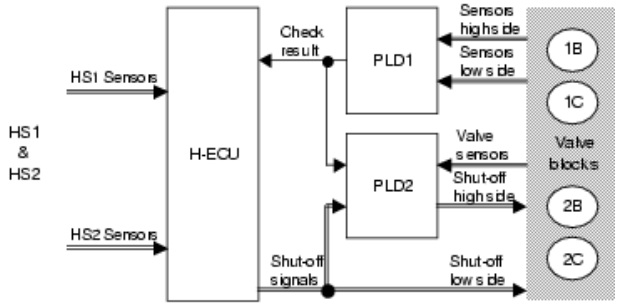

Fig. 2. The hydraulic leakage detection system.

5.3 Analysis of fault tolerance

Modules: $PLD1$, $PLD2$ and $HECU$ are represented as synchronous modules.
Fault modes: A set of fault modes F_{PLD1}, F_{PLD2} and F_{HECU} for each component has been identified. Every input to the components has been analysed and the possible faults have been modelled as corresponding fault modes.
Safety interface generation: The least restrictive environments E^{φ}_{PLD1}, E^{φ}_{PLD2}, E^{φ}_{HECU} of the components were generated by the algorithm of Section 3.1 using a SAT-based model checker (Prover plugin of the Esterel environment).

The least restrictive environment E^{φ}_{PLD1} of $PLD1$ that makes the system satisfy φ' leaves all the inputs to $PLD1$ unconstrained. By Prop. 2, $PLD1$ in the environment E^{φ}_{PLD1} is also resilient to all faults in F_{PLD1}. Analysis shows that due to their fault-tolerant design, $HECU$ and $PLD2$ satisfy the property φ' with no constraints on their environment whatsoever, i.e., $E^{\varphi}_{PLD2} = E^{\varphi}_{HECU} = True$.

Since none of $E^{\varphi}_{PLD1}, E^{\varphi}_{PLD2}$ and E^{φ}_{HECU} constrain any of the input variables of their corresponding component, these components are resilient to all single faults. Hence, the single fault resilience set of each safety interface will contain every fault mode in the corresponding fault mode set. The generated minimal environments also show that the components are resilient to all double faults, creating a safety interface that includes all pairs of faults in the double fault resilience portion of the safety interface.

Single-component faults: After computing the safety interfaces for the three components in the application (w.r.t. single and double faults), the single component fault analysis becomes trivial. No single or double fault of a single component will cause a threat to system-level safety, since all faults are included in the single fault resilience portion and all pairs of faults are included in the double fault resilience portion of the safety interface.

Multiple-component faults: By checking $\forall j \ M_i \parallel F_k \circ E_i \leq E_j$ for all module-fault pairs (M_i, F_k) where $M_i \in \{PLD1, PLD2, HECU\}$ and $F_k \in F_{PLD1} \cup F_{PLD2} \cup F_{HECU}$ we could conclude that no double fault on input signals would make a threat to system-level safety.

5.4 Results

By generating safety interfaces as described in Section 3 and using the compositional techniques of Section 4 on the aerospace application we concluded that:

- All components in the system are resilient to single faults with respect to the system level safety property φ'.
- All components in the system are resilient to double input value faults with respect to the system level safety property φ'.
- No pair of faults in the system are a threat to system level safety.
- By analysing the components individually and generating the safety interfaces using Propositions 2-4, we were able to perform the fault tolerance analysis *without* composing the whole system.

6 Conclusions

This paper extends component-based development, which has so far focused on efficient system development, to efficient analysis of safety. Certification of safety-critical systems includes providing evidence that a system satisfies certain properties even in presence of undesired faults. This process is especially costly since it has to be repeated for every component change in a system with a long life cycle. We have provided formal models and methods to support this process while hopefully reducing the burden of proof on the system integrator.

Any system will break if there are sufficient numbers of faults in its components at run-time, either due to environment effects or due to inconsistencies in designing interfaces. Safety analyses for industrial products typically assume a number of independent faults and consider the effects of single and potentially double faults. Triple and higher number of faults are typically shown to

be unlikely and not studied routinely. Our component interfaces capture what an integrator can assume about the resilience that a component offers with respect to single and double faults. The model could be extended to multiple faults (triple and more) but then the combinatoric complexity would hamper the automatic support for formal analysis. Already with this granularity, we believe that there are enough gains in efficiency for the analysis performed at system level.

This paper uses a general fault model that covers arbitrary value changes at component inputs. While this model is a powerful, for some cases it may be too general and modelling specific faulty behaviour may improve analysis efficiency. The use of reactive modules as generic base allows for more specific models in future studies, such as handling transient faults with given behaviours.

The entire approach has so far had a qualitative nature. Many safety-critical systems have to estimate a quantitative (probabilistic) measure of reliance on a particular analysis. The study of the extensions of this model to quantitative analyses is a topic for future work.

We have assumed that the misbehaviour of a component's environment can be captured by a discrete model (based on value domains with finite range). An extension could consider more specific fault modes arising from interactions with a physical environment given a continuous model. Also common mode failures were excluded at this stage. Another future direction is efficient generation of environment models. The naive approach presented here can be used as a proof of concept and can obviously be improved by more advanced techniques.

Acknowledgements This work was supported by the Swedish strategic research foundation project (SSF), and the national aerospace research program NFFP. Many thanks are due to our industrial partners from Saab AB for numerous discussions.

References

1. M. Abadi and L. Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132, 1993.
2. O. Åkerlund, S. Nadjm-Tehrani, and G. Stålmarck. Integration of formal methods into system safety and reliability analysis. In *Proceedings of* 17^{th} *International Systems Safety Conference*, pp. 326–336, 1999.
3. R. Alur and T. A. Henzinger. Reactive modules. In *Proc.* 11^{th} *Symposium on Logic in Computer Science*, pp. 207–218. IEEE Computer Society, 1996.
4. U. Aßman. *Invasive Software Composition*. Springer Verlag, 2003.
5. G. Berry. *The Esterel v5 Language Primer*. CMA, Sophia Antipolis, 2000.
6. A. Bondavalli and L. Simoncini. Failures classification with respect to detection. In *Proc. of* 2^{nd} *IEEE Workshop on Future Trends in Distributed Computing Systems*, pp. 47–53. IEEE Computer Society, 1990.
7. M. Bozzano and et al. ESACS: an integrated methodology for design and safety analysis of complex systems. In *ESREL 2003*, pp. 237–245. Balkema, June 2003.
8. A. Burns and A. J. Wellings. HRT-HOOD: a structured design method for hard real-time systems. *Real-Time Systems*, 6(1):73–114, 1994.
9. I. Crnkovic, J. Stafford, H. Schmidt, and K. Wallnau, editors. *Proc. of 7th Int. Symposium on Component-Based Software Engineering*, LNCS 3054. Springer, 2004.

10. F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, editors. 3^{rd} *Int. Symp. on Formal Methods for Components and Objects*, LNCS. Springer, 2004.
11. J. Elmqvist and S. Nadjm-Tehrani. Intents and upgrades in component-based high-assurance systems. In *Model-driven Software Development*. Springer, 2005.
12. P. Fenelon, J. A. McDermid, M. Nicolson, and D. J. Pumfrey. Towards integrated safety analysis and design. *SIGAPP Applied Computing Review*, 2(1):21–32, 1994.
13. L. Grunske, B. Kaiser, and R. H. Ruessner. Specification and evaluation of safety properties in a component-based software engineering process. In *Embedded system development with components*. Springer Verlag, 2005. to appear.
14. N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In *Proc. AMAST'93*, pp. 83–96. Springer, 1994.
15. J. Hammarberg and S. Nadjm-Tehrani. Formal verification of fault tolerance in safety-critical configurable modules. *International Journal of Software Tools for Technology Transfer*, Online First Issue, Dec. 14, 2004. Springer Verlag.
16. E. Henley and H. Kumamoto. *Reliability Engineering and Risk Assessment*. Prentice Hall, 1981.
17. T. A. Henzinger, S. Qadeer, S. K. Rajamani, and S. Taşiran. An assume-guarantee rule for checking simulation. In *Proc.* 2^{nd} *Int. Conf. FMCAD*, pp. 421–432, 1998.
18. C. B. Jones. *Development Methods for Computer Programs Including a Notion of Interference*. PhD thesis, Oxford University, 1981.
19. J. Jürjens. Developing safety-critical systems with UML. In *Proc.* 6^{th} *Int. Conf. UML 2003*, LNCS 2863, pp. 360–372. Springer, 2003.
20. J. H. Lala and R. E. Harper. Architectural principles for safety-critical real-time applications. *Proc. of the IEEE*, 82(1):25–40, Jan. 1994.
21. H. C. Li, S. Krishnamurthi, and K. Fisler. Interfaces for modular feature verification. In *Proc. of the* 17^{th} *IEEE Int. Conf. on Automated Software Engineering*, pp. 195–204. IEEE Computer Society, 2002.
22. Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems: Specification*. Springer-Verlag, 1992.
23. J. Misra and K. M. Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, 7(4):417–426, 1981.
24. Y. Papadopoulos, J. A. McDermid, R. Sasse, and G. Heiner. 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.
25. D. J. Pumfrey. *The Principled Design of Computer System Safety Analyses*. PhD thesis, Department of Computer Science, University of York, 2000.
26. A. Rauzy. Mode automata and their compilation into fault trees. *Reliability Engineering and System Safety*, 78:1–12, 2002.
27. J. A. Stankovic. Vest - a toolset for constructing and analyzing component based embedded systems. In *Proc. EMSOFT'01*, LNCS 2211, pp. 390–402. Springer, 2001.
28. D. B. Stewart, R. A. Volpe, and P. K. Khosla. Design of dynamically reconfigurable real-time software using port-based objects. *IEEE Transactions on Software Engineering*, 23(12):759–776, 1997.
29. E. A. Strunk and J. C. Knight. Assured reconfiguration of embedded real-time software. In *Proc. International Conference on Dependable Systems and Networks*, pp. 367–376. IEEE Computer Society, 2004.
30. C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 2nd edition, 2002.
31. A. Tesanovic, S. Nadjm-Tehrani, and J. Hansson. Modular verification of reconfigurable components. In *Embedded System Development with Components*. Springer, 2005. to appear.