Safety-Oriented Design of Component Assemblies using Safety Interfaces

Jonas Elmqvist and Simin Nadjm-Tehrani[1]

Department of Computer and Information Science
Linköping University
Linköping, Sweden

Abstract

This paper promotes compositional reasoning in the context of safety-critical systems, and demonstrates a safety-oriented component model using an application from the automotive industry: an Adaptive Cruise Controller (ACC). The application consists of four components for which a set of 18 fault modes have been identified. We show the impact of all single faults and double faults selected from this set, on a safety property associated with the ACC assembly. Analysis related to each fault mode is performed using compositional rules and derived safety interfaces for each component.

The derivation of safety interfaces for the ACC components has been supported by implementation of two extensions to the SCADE tool set: (1) a front end that iteratively and automatically builds the environment in which the component is resilient in presence of a given fault, (2) fault mode libraries that can be reused for modeling several classes of faults affecting the input of a component. The result of the study is the illustration of system level safety in presence of certain single and double faults, based on compositional reasoning and the automatically generated interfaces. The component model uses reactive modules as the formal notation. The instantiation of the model in terms of modules specified in SCADE provides a link between formal analysis of components in safety-critical systems and the traditional engineering processes supported by model-based development.

Keywords: Component-based system development, safety, component assemblies, safety interfaces, fault modes, SCADE

1 Introduction

Component-based software development [32,8] (CBSD) has emerged as a promising approach for developing complex software systems by composing smaller independently developed components into larger component assemblies. This approach offers means to increase software reuse, achieve higher flexibility and shorter time-to-market by the use of off-the-shelf components (COTS). However, the use of COTS in safety-critical system is highly unexplored.

First, system safety engineers need to present arguments that justify dependability of the system, based on any information available from the COTS. Typically

[1] Email: {jonel,simin}@ida.liu.se

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

COTS are not developed with generation of such arguments to certification authorities in mind. Specially, whether or not fault tolerance in components affects system properties, is not currently part of component interfaces. Secondly, the process that leads to this assurance needs to be rigorous, efficient, and easily applicable for upgrades at a later stage of the system life cycle. Thus, support for rigorous, compositional analysis from components to assemblies, and with specific focus on faults that can jeopardize system safety is a relevant area of work [16,26].

Following the trend of software components in system development, the use of critical software in complex safety-critical systems has increased. Since safety has to be addressed at the system level, an overall analysis of the component assembly is necessary. This implies that effect of possible component failures must be analysable as foreseen hazards at the system level. Also, for efficiency, there is a desire not only to reuse components, but also to reuse analysis results upon future upgrades. The use of formal models and analyses is generally accepted as a means of rigorous safety assessment in software [33,31]. However, as of now, no industrial tools exist for assessing dependability in a system built from components [12].

During recent years, a wide range of models and methods for developing systems from components have emerged [6,11]. In particular, an analytical component model has been presented that provides the means for reasoning about system safety, by reasoning about known component behaviour in presence of specified faults (i.e. safety interfaces [13]). This work is built on the component model presented therein. By generating safety interfaces of components, the component developer may specify the effects and the assumptions needed for the component to be resilient to specific faults. This characterises how a component might contribute towards jeopardizing a given safety property at system level [13]. The framework has so far been applied to an aerospace application with only Boolean variables in the Esterel Studio tool set.

In this paper, we apply the proposed technique to a non-trivial case study, namely an automotive Adaptive Cruise Controller (ACC) with non-Boolean variables. In order to cope with the analysis on integer functions we utilise the tool set SCADE [15]. A methodology for building fault mode libraries in order to use this method efficiently is illustrated. We have also implemented parts of the framework as a front-end to SCADE for generating safety interfaces. This application illustrates the possibility of reuse of previous partial analysis results in future upgrade scenarios.

The paper is structured as follows. In section 2 we present some basic definitions and recall our earlier work upon which the rest of the paper builds. In section 3, an overview of the ACC case study is given. Section 4 presents the methodology for use of fault mode libraries and the front-end to SCADE. In sections 5 and 6, the activities seen from the two perspectives (system developer and component developer) are described and the result of the analysis is presented. Section 7 presents related work while section 8 concludes the paper.

2 Preliminaries

Our general formalism for modules is a special class of *reactive modules* [2] with synchronous composition, finite variable domains and non-blocking transitions, that we call synchronous modules (from now on simply called modules).

Definition 2.1 [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*.

A *state* q of a module M is an interpretation of the variables in V and the successor of a state is obtained at each transition by updating the controlled variables of the module. This is essentially the definition of a module's function as a transition system. The execution of a module produces a state sequence $\bar{q} = q_0 \ldots q_n$. A trace $\bar{\sigma}$ is the corresponding sequence of observations on $\bar{q}$, with $\bar{\sigma} = q_0[V_{obs}] \ldots q_n[V_{obs}]$, where $q[V']$ is the projection of q onto a set of variables $V' \subseteq V$. The trace language of M, denoted $\mathcal{L}_M$, is the set of traces of M. A property φ, a set of traces on $V' \subseteq V$, is a *safety property* iff $\bar{\sigma} \in \varphi \Leftrightarrow \bar{\sigma}' \in \varphi$ for any finite prefix $\bar{\sigma}'$ of $\bar{\sigma}$.

Definition 2.2 [Model] A module M *models* a property φ, denoted $M \models \varphi$, iff every trace of M belongs to the traces of φ.

Definition 2.3 [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 $(q, i, q') \in \delta$ iff $(q[V_{ctrl}^M], (i \cup q)[V_i^M], q'[V_{ctrl}^M]) \in \delta^M$ and $(q[V_{ctrl}^N], (i \cup q)[V_i^N], q'[V_{ctrl}^N]) \in \delta^N$.

Reactive modules can be related via trace semantics: a module M refines module N if all possible traces of M also are possible traces of N.

Definition 2.4 [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 $M \preceq N$, if (1) $V_o^N \subseteq V_o^M$, (2) $V_i^N \subseteq V_{obs}^M$ and (3) $\{\bar{\sigma}[V_{obs}^N] : \bar{\sigma} \in \mathcal{L}_M\} \subseteq \mathcal{L}_N$.

To be able to apply formal analysis of the behaviour of a component in presence of faults in its environment, we need to define a formal model of the faults.

Definition 2.5 [Input Fault Mode] An input fault mode F_k of a module M is a module with one input variable $v_k^f \notin V^M$ and one output variable $v_k \in V_i^M$, both of the same type D_k.

Faults in the environment of a component are modelled as faulty inputs to the component, and each such faulty input creates a *fault mode* for the component. The input fault of one component thereby captures the output fault of a component connecting to it. By using modules as means of modelling fault modes, the fault modelling is only limited by the expressiveness of modules. Various fault mode classes can be derived by specifying the corresponding fault modules. In traditional safety analysis, faults can be classified into the following high-level categories: *omission faults*, *value faults* (coarse and subtle), *commission faults* and *timing faults* [4,13]. In this work, we do not focus on timing faults and our work does not include the process of determining fault modes, and hazard analysis, which is itself a different research topic. Fault modes are assumed as given, and the result of the analysis is knowledge about the potential impact of a given fault on a component assembly.

Definition 2.6 Let E be a module with $v_k \in V_o^E$ and F_k be a fault mode with input v_k^f and output v_k. We denote $E \circ F_k = E[v_k/v_k^f] \parallel F_k$ where $E[v_k/v_k^f]$ is the module E with the substitution v_k^f for v_k.

Consider a module M, an environment E and a fault mode F_k that affects the input v_k from E to M. In the resulting faulty environment $E \circ F_k$, the original output v_k of E becomes the input v_k^f of F_k, which produces the faulty output v_k as input to M. Thus, $E \circ F_k$ has the same number of observable variables as E and can be composed with M.

Now, we can define a *safety interface*.

Definition 2.7 [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 $\langle \mathsf{single}, \mathsf{double}, E^\varphi \rangle$ where

- $\mathsf{single} = \{\langle F_1^s, A_1^s \rangle, \ldots, \langle F_n^s, A_n^s \rangle\}$ where $F_j^s \in F$ and A_j^s is a module composable with M, such that $M \parallel (A_j^s \circ F_j^s) \models \varphi$
- $\mathsf{double} = \{\langle F_1^d, A_1^d \rangle, \ldots, \langle F_n^d, A_n^d \rangle\}$ with $F_k^d = \{\langle F_k^1, F_k^2 \rangle \mid F_k^1, F_k^2 \in F,\ F_k^1 \neq F_k^2\}$ such that $M \parallel (A_k^d \circ (F_k^1 \parallel F_k^2)) \models \varphi$
- E^φ is an environment in which $M \parallel E^\varphi \models \varphi$.

The safety interface makes explicit which single and double faults the component tolerates when placed in a specific environment (abstracted by the element A in the tuples above). These environments can be seen as requirements that the component places on its environment in order to be resilient to the declared faults with respect to the safety property φ. In particular, E^φ is an environment abstraction in which M functions if there are no faults. Similarly, A_j^s and A_k^d are environment abstractions in which the component is resilient to the specific single fault F_j and double fault F_k respectively. Note that the safety interface does not have to treat all faults in F (and in fact could be empty), meaning that the component developer only specifies what is known about the component in presence of faults.

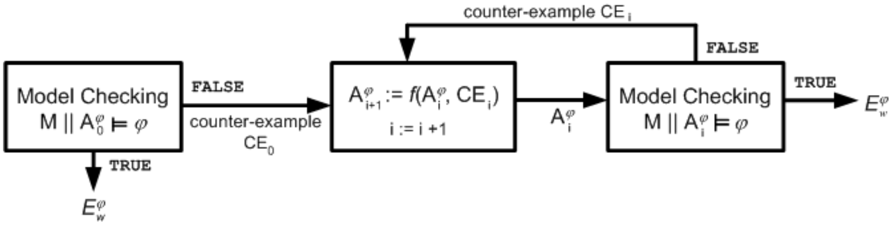

Fig. 1. The environment abstraction generation algorithm (EAG-algorithm).

Definition 2.8 [Component] Let M be a module and SI^φ a safety interface for M. A component C is defined as the tuple $\langle M, SI^\varphi \rangle$.

2.1 Environment Abstraction Generation Algorithm

Creating the safety interface is not a trivial task. Major element of a safety interface are the environment abstractions. We now recall the environment abstraction generation algorithm (EAG, see Figure 1) that generates these environment abstractions.

Let M be a module, φ be a safety property and A_0 an unconstrained environment. Then the least restrictive (weakest) environment E_w^φ that composed with M satisfies φ can be generated as demonstrated in Figure 1. The algorithm uses a model checker to check whether the module M in parallel with the environment A^φ satisfies the safety property φ: $M \parallel A^\varphi \models \varphi$.

Initially, the algorithm starts out with an empty constraint A_0^φ on the environment. At each iteration i, the algorithm strengthens the constraints A_i^φ by analysing the counter-example generated by the model checker and removing the forbidden traces (represented by f in Figure 1). This corresponds to removing behaviours from (or strengthening) the environment. In the next iteration, the environment A_{i+1}^φ should at least not exhibit the behaviour reflected by the counter-example at iteration i. The infinite value range of variables creates an infinite state space. By bounding the value range we can create a finite number of states. Thus, the algorithm will terminate and stops at a fixpoint when $A_{i+1}^\varphi = A_i^\varphi = E_w^\varphi$. As usual, there is the risk of combinatorial explosion, but the treatment of that is a separate research topic on improvements of this naive algorithm.

As mentioned above, a safety interface also expresses how a module behaves in presence of specific faults; in particular in which environment the module tolerates a given fault. The EAG-algorithm can also be used to derive these environment abstractions. Input to the algorithm is now M and $E_w^\varphi \circ F_k$, i.e. the faulty environment. The output A_k, is the environment abstraction in which the module M tolerates F_k. Two cases may occur during the above process:

- **Unconstrained environment:** Those cases when a component itself satisfies the safety property for all inputs to the component, i.e. $M \models \varphi$. This implies that the component tolerates the given fault in all possible environments.
- **Contrained environment:** The normal case when the EAG-algorithm terminates with a constrained environment. The generated environment abstraction together with the specific fault is added to the safety interface. The special case

when the generated environment abstraction has no traces indicates that the component is not tolerant of this fault no matter in which environment it is placed. This is a valuable knowledge for the component developer.

A special class of constrained environments are those in which $\forall v_k \in V_o^M, v_k \notin V_i^{\varphi}$, which practically means that the safety property is not directly affected by the outputs of this component. For those cases, the abstraction trivially becomes $A_k^s = \varphi$ and terminates immediately.

3 Case study: Adaptive Cruise Control

In this section we introduce an automotive application to illustrate our methodology; the Adaptive Cruise Control (ACC) (informally described in [21]). Beside its safety and real-time aspects, the case study is particularly interesting because structuring and reuse is of importance in competitive industries today, especially the automotive industry. However, we are well aware that such formal safety analysis is only a small part of the puzzle that corresponds to developing complex automotive electronics in a competitive market.

3.1 General description and safety requirements

The ACC is an extension of the conventional in-vehicle cruise control function that can be found in most cars today. As well as the traditional functionality of a cruise control, i.e. adapting the vehicle to a specific speed set by the driver, the ACC may also adapt the distance to a vehicle in front or adapt to the current speed limit of the specific road section.

If the ACC application discovers a target vehicle in front of the own vehicle, the ACC will adapt the speed of the own vehicle to ensure a safe distance to the target vehicle. If the target vehicle disappeares, the ACC will work as a conventional cruise control. For safety reasons, the driver should at any time be able to take control of the car by braking or using the throttle.

3.2 ACC architectural decomposition

To reduce the complexity and to apply a component-based approach to this case study, the functionality of the ACC is divided into the following four components (see Figure 2):

- **Speed Limit Component** Calculates and controls the speed that the ACC should adapt. The driver sets a maximum speed limit which the vehicle should not exceed when the ACC is activated.
- **Object Recognition Component** Responsible for detecting when vehicles appear within a fixed distance in front of the car.
- **Mode Switch** Controls the mode of the ACC, i.e. whether the ACC is in STANDBY, ON or OFF mode depending on the inputs from the driver and the current speed of the car.
- **ACC Controller Component** Handles the adaptation to speed or distance using two different controllers, one for speed control and one for distance control.

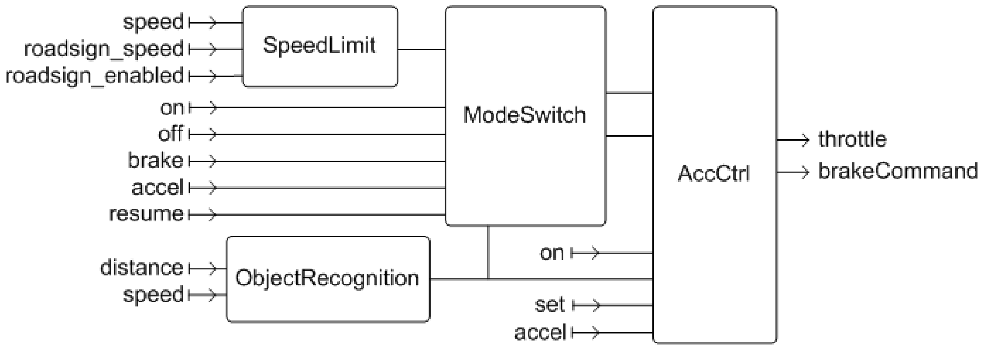

Fig. 2. The ACC architecture.

Components communicate through typed signals that are either Boolean or integers. In some cases, Boolean variables may also be triggering signals that trigger the execution of the connected component.

3.3 Formalizing Safety Properties

When introducing an in-vehicle function into a subsystem such as the ACC, system engineers need to verify that safety properties are assured and all new hazards that are introduced into the system are carefully analysed. In our exposure of the case study in this paper, we will consider and analyse the following safety property for the ACC:

φ : *When the ACC is in* ACC-Mode*, the speed is higher than 50 km/h, and there is a vehicle closer than 50 meters in front, the system should not accelerate.*

3.4 Fault modes

A part of the preliminary safety assessment procedure is to identify, as far as possible, faults in the system and evaluate their impact as described in Section 2. For example, each triggering signal can be effected by an *omission* or *commission fault*. Every variable can be effected by a *value fault*, i.e. an unintended change in the value of the signal. More specifically, signals can get stuck at a certain value giving rise to a *StuckAt*-fault.

Table 1 shows a selection of fault modes that were identified in the ACC system and referred to in the forthcoming section. Other fault modes can be handled analogously, but are not treated in the following sections.

4 Tool support for deriving Safety Interfaces

Existing tools with an ambition to support component-based development are mainly based on UML 2.0 which is promoted for representing and describing (software) components. However, none of these tools supports automatic derivation of any kind of interfaces. This section introduces an overview of our safety analysis methodology and presents an approach to aid the component developer in this process.

Fault	*Type*	*Component*	*Input*	*Fault*	*Type*	*Component*	*Input*
F_1	StuckAt	ModeSwitch	On	F_{11}	StuckAt	AccCtrl	RegulON
F_2	StuckAt	ModeSwitch	Off	F_{12}	StuckAt	AccCtrl	RegulOFF
F_3	Value	ModeSwitch	SpeedOK	F_{13}	StuckAt	AccCtrl	ACCMode
F_4	StuckAt	ModeSwitch	Resume	F_{14}	Value	AccCtrl	Accel
F_5	Value	ModeSwitch	Brake	F_{15}	Commission	AccCtrl	RegulON
F_6	Value	ModeSwitch	Accel	F_{16}	Omission	AccCtrl	RegulON
F_7	StuckAt	AccCtrl	On	F_{17}	StuckAt	SpeedLimit	RSEnabled
F_8	StuckAt	AccCtrl	Set	F_{18}	Value	SpeedLimit	Speed
F_9	Value	AccCtrl	Speed	F_{19}	Value	SpeedLimit	RoadSign
F_{10}	Value	AccCtrl	Distance	F_{20}	Value	ObjectRecognition	Distance

Table 1
Identified possible faults in the system.

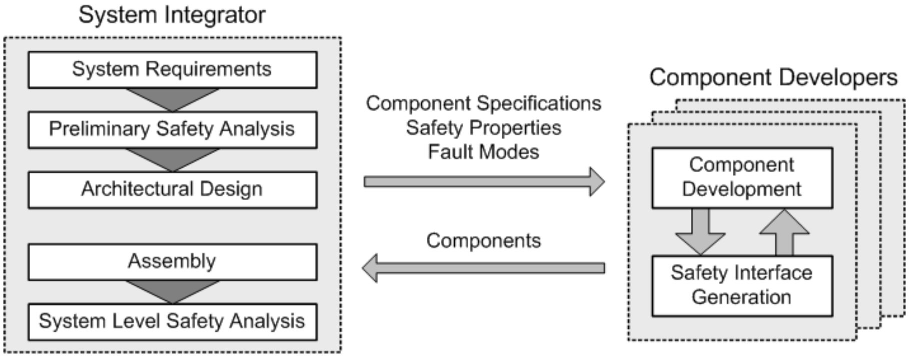

Fig. 3. The component-based system development process.

4.1 Safety Analysis Methodology

The idea behind component-based system development is to divide the system development between two parties: the system integrator and the component developers. The responsibility of the component developers is to design and implement components according to the specification given by the system integrators. Adopting a component-based approach to the development of safety-critical systems is not trivial since safety is a system level attribute. Thus, it is not possible to outsource the complete safety assessment process to the component developers. Hence, traditionally, safety analysis is done at the system level by the system safety engineers during system integration. Our methodology of component-based safety analysis based on safety interfaces divides the effort of safety analysis between the above two parties; the system integrator and the component developer as depicted in Figure 3.

In order to make the system-level safety analysis possible, the component developers must supply the system integrators with a model of a component including a safety interface. One method for generating the safety interface was introduced in Section 2. However, the iterative generation of the environment abstraction is in practice too tedious to do manually since it may involve placing a very large number of constraints on the environment. To make the process of generating safety interfaces efficient, some tools are necessary to aid the component developers in this process. Several tools are being extended to handle component (interaction or functional) interfaces , e.g. Autofocus [22], Matlab [24]. Our focus here is the interfaces specific to fault tolerance and safety.

4.2 Front-end to SCADE

In this study, the development environment SCADE and the built-in Design Verifier[2] [14] was used. By modelling the system in SCADE, and specifying the safety properties in the underlying language Lustre [19], the Design Verifier is used to verify whether a certain safety property holds in the system or not.

If the system violates the property, the Design Verifier has found a bad state and terminates the analysis. Output from the analysis is the trace (i.e. sequence of variable assignments[3]) that leads to that specific bad state, i.e. a counter-example. Thus, SCADE and the Design Verifier can help to implement the algorithm for generating the environment abstraction described in Section 2.1.

To make the application of the EAG-algorithm more practical in large state spaces, a front-end to SCADE was implemented to automate the procedure. By automatic analysis of the counter-examples that the Design Verifier returns, the environment of each component may be constrained using *assertions* in Lustre.

For example, if the environment consists of the signals `On, Off`, and `Resume` and the counter example returns the following trace

```
Step 0: On=false, Off=false, Resume=true;
Step 1: On=true, Off=true, Resume=false;
```

that violates the safety property, we may use the following assertion construction to remove this trace during the next iteration.

```
assert not (Pre(On)=false and Pre(Off)=false and Pre(Resume)=true
                      and (On=true and Off=true and Resume=false);
```

Figure 4 (a) depicts the application of the EAG-algorithm within the SCADE environment, and the role of our implemented front-end. The initial call to the model checker is done with an unconstrained environment A_0^φ and new environments A_i^φ are generated by analysing counter examples and creating constraints. By adding these constraints to the environment of the component, and running the model checker again with the new environment A_{i+1}^φ, the bad behaviour from the iterations is removed from the environment. Finally, all bad behaviours of the environment with respect to the property φ will be removed and the algorithm will terminate with an environment abstraction (E_w^φ) constrained enough to make the safety property valid when composed with the module in question.

[2] A SAT-based model checker.

[3] The Design Verifier only present variable assignment of signals that are critical to the falsification of the safety property and omits others (i.e. *don't care*)[14].

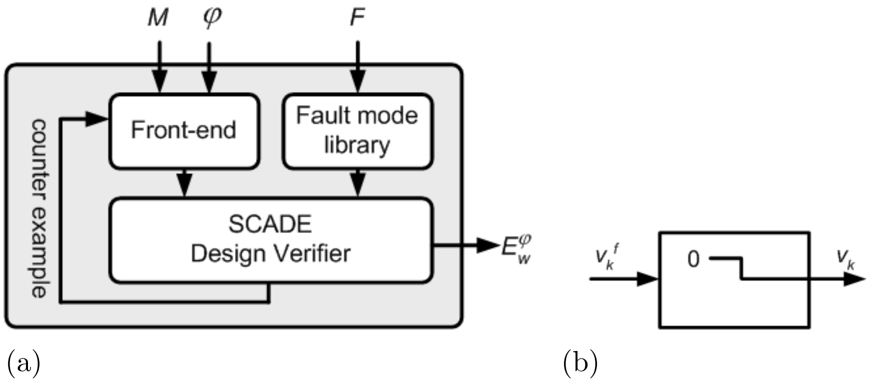

Fig. 4. a) The SCADE front-end. b) Example of fault mode (*StuckAt*)

4.3 Fault mode library

As mentioned in Section 2, faults can be classified into a set of high-level fault modes. In order to create a safety interface of a component, environment abstractions must be generated for each fault that the component is resilient to. To make this procedure more efficient for the component developer, a *fault library* may be used. The SCADE environment allows to store and import models as libraries to enable reuse. By modelling the high-level fault modes, e.g. a *StuckAt*-fault (see Figure 4 (b)), and importing them into the component analysis process, much work can be reused and time can be saved, as shown later on in the example (see Section 6.1).

5 Safety Interfaces for ACC components

The whole ACC system and its four components were divided into 12 SCADE nodes which can be translated into over 2000 lines of automatically generated C code. The front-end to SCADE and the fault mode library were both used to generate safety interfaces for the four ACC components. In this section, we will present the development and the safety interface generation of the *ModeSwitch* component.

5.1 ModeSwitch

An initial design was first created based on the functional requirements derived in the initial phase of the development process. When a satisfactory functional design M was implemented in SCADE for the *ModeSwitch* component, the next step in the process was to generate the safety interface of the component. As depicted in Figure 4, input to this process is the module M, the safety property φ (from Section 3.3) and the set of faults F (from Table 1). The safety interface $SI^{\varphi}{}_{ModeSwitch}$ consists of three elements: the two tuples `single` and `double` and the environment E^{φ}. First of all, the EAG-algorithm was used to generate the environment E^{φ} by using the front-end to SCADE. The tool terminated with an environment E^{φ} with 232 unique constraints on the output variables of E^{φ}. Note that the EAG-algorithm does not only lead to some matching environment. It may also present weaknesses in the component design to the component developer. By analysing the constraints generated by the process, design flaws may be found. For example, while generating E^{φ} for *ModeSwitch*, 2 major and 4 minor design flaws were caught and corrected during the process. As an example, one mode switch from ACC_ON to ACC_OFF was omitted in the initial design, but this error was fixed after the first run of the EAG-algorithm. This presents an added value to the component developer.

For each single fault F_j in Table 1 affecting *ModeSwitch*, the front-end to SCADE was used to generate the specific environment A^s_j. Each combination of double faults that affects only *ModeSwitch* was also considered. The final result of the analysis was that this component tolerates all single faults but not in all environments. For example, a *StuckAtTrue*-fault affecting the signal `On` (i.e fault F_1) generates an environment with 17 constraints.

All other components were designed in a similar fashion and the safety interfaces for the components were all generated using the front-end to SCADE.

6 Safety of ACC assembly

For complex systems such as the ACC, it is difficult to reason about its impact on safety, especially in presence of faults in the multiple (upgraded) components. Assume-guarantee reasoning is one method for handling manageability and scalability of complex systems. This section presents a technique that enables the system integrator to perform system level safety analysis based on component models.

6.1 System Level Safety Analysis

The goal of the system integrator when analysing the assembly of components is to determine whether the whole assembly satisfies the specific safety properties in presence of faults, i.e. for a fault $F_j \in F$ effecting component M_j, check if:

$$M_1 \parallel M_2 \parallel \ldots \circ F_j \parallel M_j \parallel \ldots \parallel M_n \models \varphi \tag{1}$$

However, composing all the components may be infeasible for many reasons, both complexity and lack of efficiency during upgrades. Upgrades typically affect a local part of a complex design. The idea is to avoid performing global analysis on all unaffected parts. Instead, we apply an n-module circular assume-guarantee rule that can be found in [13]. By applying this rule, the system integrator may investigate Equation 1 without paying the price of an expensive overall composition. Reasoning is restricted to changes affected by the upgraded module and without having to redo the entire analysis each time a component changes.

The system integrator checks whether the fault F_j is in the safety interface for any component that has an input affected by F_j. All single faults in F that do not appear in `single` in the safety interface of the relevant components will actually be a threat to the overall system safety in relation to the safety property under study.

In the case of the ACC, we present here the application of the rule to the 4-component assembly in presence of fault F_1. Let M_M,M_A,M_S,M_O represent the *ModeSwitch*, the *AccCtrl*, the *SpeedLimit*, and the *ObjectRecognition* modules respectively. Analogously, E^φ_A, E^φ_S, E^φ_O denote the environments that are provided in the safety interfaces associated with M_A, M_S and M_O respectively. Then:

$$\begin{array}{rlrl}
M_M \parallel A_1^s \circ F_1 & \models \varphi & M_M \parallel A_1^s \circ F_1 & \preceq E_A^\varphi \\
 & & M_M \parallel A_1^s \circ F_1 & \preceq E_S^\varphi \\
 & & M_M \parallel A_1^s \circ F_1 & \preceq E_O^\varphi \\
M_A \parallel E_A^\varphi & \models \varphi & M_A \parallel E_A^\varphi & \preceq A_1^s \circ F_1 \\
 & & M_A \parallel E_A^\varphi & \preceq E_S^\varphi \\
 & & M_A \parallel E_A^\varphi & \preceq E_O^\varphi \\
M_S \parallel E_S^\varphi & \models \varphi & M_S \parallel E_S^\varphi & \preceq A_1^s \circ F_1 \\
 & & M_S \parallel E_S^\varphi & \preceq E_A^\varphi \\
 & & M_S \parallel E_S^\varphi & \preceq E_O^\varphi \\
M_O \parallel E_O^\varphi & \models \varphi & M_O \parallel E_O^\varphi & \preceq A_1^s \circ F_1 \\
 & & M_O \parallel E_O^\varphi & \preceq E_A^\varphi \\
 & & M_O \parallel E_O^\varphi & \preceq E_S^\varphi \\
\hline
\multicolumn{4}{c}{M_M \parallel (M_A \parallel M_S \parallel M_O) \circ F_1 \models \varphi}
\end{array}$$

The first premise in the left column above is trivially true if $\langle F_1, A_1^s \rangle \in$ single in the safety interface of $ModeSwitch$. The three other premises in the left column are trivially true based on the definition of safety interfaces. Checking the rest of the premises above is possible using the Design Verifier, pair-wise composing the constraint on the environment with the modules and launching the model checker. Also, as seen above, many of these checks will be needed more than once which means that the result of the analysis of the first fault $F_1 \in F$ can be reused when analysing other faults in F. For example, when analysing the effect of fault F_2 that also affects M_M, only 6 out of the 12 premises on the right column are needed. The result of the other 6 premises can be reused from earlier analysis.

If all premises are true, we can conclude that the system will tolerate the specific single fault. If one or more of the premises do not hold, we can conclude that this fault is a threat to overall system safety of the assembly.

When two single faults appear simultaneously, they create a double fault. If both of these faults affect the same component, then the system level analysis of this double fault is handled analogously to single faults, except that the `double` element is used instead of `single`. However, when the individuals in a double fault affect two *different* components, the safety analysis process becomes a bit more complex.

Consider a double fault $\langle F_1, F_7 \rangle$ where the individual faults according to Table 1 affect *ModeSwith* and *AccCtrl*. In this case, the premises that are needed to prove resilience to these faults have to be changed in accordance with environment abstractions in the safety interfaces of the effected components. Each E_A has to be replaced with $A_7^s \circ F_7$ in order to achieve the correct result. In a similar fashion as with single faults, if one or more of the premises are falsified, then the system does not tolerate this specific double fault.

6.2 Result

The result of the safety analysis for the whole system, using all faults in Table 1 shows that:

- the ACC assembly is only resilient to 8 single faults, F_1, F_3, F_4, F_7, F_8, F_{12}, F_{16} and F_{19}.
- the ACC assembly is resilient to two double faults $\langle F_1, F_4\rangle$ and $\langle F_4, F_8\rangle$.

These faults can individually or in pairs, e.g. $\langle F_1, F_4\rangle$, appear in the system without jeopardising the overall safety with respect to the safety property φ. From now on the task of the system engineer will be focused on single faults not in the list above, and double faults that constitute a threat. The work proceeds with quantifying the risk associated with each fault (or fault combination) and providing mitigations against them.

7 Related Work

We consider related works in the following three areas in sequel: components and safety assessment, modular verification techniques, and model-based development.

Our work is rooted in earlier efforts to combine functional design and safety analysis using the same formal model [1,5,20]. Attempts to study component behaviour in this context are recent and few. Bishop et. al. [3] addressess the problem of safety of software components (COTS) by classifying the criticality of software components and by adapting HAZOP [30] to assess the safety impact of software component failures. However, their work focuses more on a high-level safety analysis and does not include any formal verification techniques. The B-method which is a formal development method used to produce industrial safety-critical software has also been applied to component based systems [28]. That work allows users to create more trustable components with the aid of formal proofs, testing and runtime checking, and also to generate target code from the component specifications. However, in the referred work, they only reason about local component properties and do not address global system properties. The idea of supplying COTS components with specific safety information has been proposed by Dawkins et al [9] but without any support for formal analysis.

UML 2.0 is promoted as a suitable language for component modelling. 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), including nodes and links [25]. This model has the benefit that it narrows the gap between a system realised as a set of functions and a system realised as a set of components (by adopting UML-based notation). The model has been described in a formal notation that has a potential for connection to formal verification tools. Other formal component models include the SaveCCM that so far focuses on timing properties [6].

Using modular verification techniques within component assemblies is an active area of work. For example, in [23] system properties are proved by independent model-checking of a group of small state spaces with the help of interface automata [10]. However, their work focuses on communication protocols while abstracting away from the data values being communicated. Similarly, Chaki et al [7] present methods for finite state abstractions of low-level C components. The approach of [17] is related to ours in terms of environment assumptions. They present a model checking algorithm for linear transition systems that returns an assumption that characterizes exactly those environments in which the component satisfies the property. Our work can be considered as extension of that work to analysis of fault tolerance by considering environment faults as input to the analysis.

Some research in the area of model-based development focus on fault modelling. For example, [24] presents a method for model-based safety analysis with fault modelling and formal verification as means for safety assessment. Their work however, does not address modular verification techniques. Other works on safety and UML includes [27] which presents methods and tools for automated safety checking in UML statecharts specifications. Grunske et al. [18] present a methodology for model-based hazard analysis for component-based software systems based on State Event Fault Trees. However, safety analysis is performed on the composed system and it requires modelling of failure behaviour and propagation inside a component. In our work, fault propagation inside the components is already captured by the formal functional model. The Altarica language [29] was designed to formally specify the behaviour of systems when faults occur. Tools such as a fault tree generator and a model checker exist for analysing Altarica models. However, components and interfaces are not considered and the language does not differentiate between transient and permanent faults which our model of faults can do.

The relation of our approach to the traditional FTA/FMEA techniques in system safety are described in earlier work [13,20].

8 Conclusions

This paper illustrates a component-based safety analysis framework with the help of formal models and tools to support the process. Our work shows promising results on an automotive application with 4 non-trivial components and 20 fault modes. We show how the component-based methodology can be cast in terms of models underlying traditional engineering processes by illustrating the extension needed to apply the method with the SCADE tool set, namely analysis front-end and fault libraries.

However, more work needs to be done towards a more efficient method of generating safety interfaces and refinement checking. Also, employing such techniques in development environments that are specifically built around the notion of components is worthwhile to pursue. More advanced methods for analysing environment constraints could also be useful to increase the possibility for component developers to find design flaws early in the development process.

Acknowledgements This work was supported by the Swedish strategic research foundation (SSF) project SAVE, and the national aerospace research program NFFP. The authors would like to thank Marius Minea and Lars Grunske for comments on earlier drafts of this paper. We would also like to thank the anonymous reviewers for their valuable input.

References

[1] Åkerlund, O., S. Nadjm-Tehrani and G. Stålmarck, *Integration of formal methods into system safety and reliability analysis*, in: *17th Int Systems Safety Conference*, 1999.

[2] Alur, R. and T. A. Henzinger, *Reactive modules*, in: *Proc. of the 11th Symposium on Logic in Computer Science* (1996), p. 207.

[3] Bishop, P. G., R. E. Bloomfield, T. Clement and S. Guerra, *Software criticality analysis of cots/soup*, in: *the 21st International Conference on Computer Safety, Reliability and Security* (2002), pp. 198–211.

[4] Bondavalli, A. and L. Simoncini, *Failure classification with respect to detection*, in: *Proceedings of 2nd IEEE Workshop on Future Trends in Distributed Computing*, Cairo, Egypt, 1990, pp. 47–53.

[5] Bozzano, M. and et al, *ESACS: an integrated methodology for design and safety analysis of complex systems*, in: *ESREL 2003* (2003), pp. 237–245.

[6] Carlson, J., J. Håkansson and P. Pettersson, *SaveCCM: An analysable component model for real-time systems*, in: *Proceedings of the 2nd Workshop on Formal Aspects of Components Software (FACS 2005)*, Electronic Notes in Theoretical Computer Science (2005).

[7] Chaki, S., E. Clarke, A. Groce, S. Jha and H. Veith, *Modular verification of software components in C*, IEEE Transactions on Software Engineering **30** (2004), pp. 388–402.

[8] Crnkovic, I., J. Stafford, H. Schmidt and K. Wallnau, editors, "Proc. of 7th Int. Symposium on Component-Based Software Engineering," Springer Verlag, 2004.

[9] Dawkins, S. and T. Kelly, *Supporting the use of cots in safety critical applications*, IEE Seminar Digests **1997** (1997), pp. 8–12.

[10] de Alfaro, L. and T. A. Henzinger, *Interface automata*, in: *Proceedings of the 8th European software engineering conference* (2001), pp. 109–120.

[11] de Boer, F. S., M. M. Bonsangue, S. Graf and W.-P. de Roever, editors, "2nd Int. Symposium on Formal Methods for Components and Objects," Springer-Verlag, 2004.

[12] Elmqvist, J. and S. Nadjm-Tehrani, "Model-driven Software Development - Volume II of Research and Practice in Software Engineering," Springer Verlag, 2005 pp. 289–304.

[13] Elmqvist, J., S. Nadjm-Tehrani and M. Minea, *Safety interfaces for component-based systems.*, in: R. Winther, B. A. Gran and G. Dahll, editors, *SAFECOMP*, Lecture Notes in Computer Science **3688** (2005), pp. 246–260.

[14] Esterel Technologies, "Design Verifier User Manual," (2004).

[15] Esterel Technologies, "SCADE 4.3.1," (2005).

[16] Fenelon, P., J. A. McDermid, M. Nicolson and D. J. Pumfrey, *Towards integrated safety analysis and design*, SIGAPP Applied Computing Review **2** (1994), pp. 21–32.

[17] Giannakopoulou, D., C. S. Pasareanu and H. Barringer, *Component verification with automatically generated assumptions*, Automated Software Engineering **12** (2005), pp. 297–320.

[18] Grunske, L., B. Kaiser and Y. Papadopoulos, *Model-driven safety evaluation with state-event-based component failure annotations*, in: G. T. Heineman, I. Crnkovic, H. W. Schmidt, J. A. Stafford, C. A. Szyperski and K. C. Wallnau, editors, *CBSE*, Lecture Notes in Computer Science **3489** (2005), pp. 33–48.

[19] Halbwachs, N., P. Caspi, P. Raymond and D. Pilaud, *The synchronous data-flow programming language LUSTRE*, Proceedings of the IEEE **79** (1991), pp. 1305–1320.

[20] Hammarberg, J. and S. Nadjm-Tehrani, *Formal verification of fault tolerance in safety-critical configurable modules*, International Journal of Software Tools for Technology Transfer (STTT) (2004), pp. 268–279.

[21] Hansson, H., M. Åkerholm, I. Crnkovic and M. Törngren, *SaveCCM - a component model for safety-critical real-time systems*, Euromicro Component Models for Dependable Systems **00** (2004), pp. 627–635.

[22] Huber, F. and B. Schätz, *Integrated development of embedded systems with autofocus.*, Technical Report TUMI-0701, Fakultat fur Informatik, TU Munchen (2001).

[23] Jin, Y., C. Lakos and R. Esser, *Component-based design and analysis: A case study.*, in: *International Conference on Software Engineering and Formal Methods* (2003), pp. 126–135.

[24] Joshi, A. and M. P. Heimdahl, *Model-based safety analysis of Simulink models using SCADE design verifier*, in: *the 21st International Conference on Computer Safety, Reliability and Security*, LNCS **3688** (2005), pp. 122–135.

[25] Jürjens, J., *Developing safety-critical systems with UML*, in: *UML 2003 - The Unified Modeling Language. Model Languages and Applications*, LNCS **2863** (2003), pp. 360–372.

[26] Kelly, T. and J. McDermid, *Safety case construction and reuse using patterns*, in: P. Daniel, editor, *the 16th International Conference on Computer Safety, Reliability and Security* (1997), pp. 55–69.

[27] Pap, Z., I. Majzik and A. Pataricza, *Checking general safety criteria on UML statecharts*, in: *SAFECOMP '01: Proceedings of the 20th International Conference on Computer Safety, Reliability and Security* (2001), pp. 46–55.

[28] Petit, D., V. Poirriez and G. Mariano, *The B-method and the component-based approach*, Journal of Integrated Design and Process Science **8** (2004), pp. 65–76.

[29] Point, G. and A. Rauzy, *Altarica - constraint automata as a description language*, European Journal on Automation **3** (1999), pp. 1033–1052.

[30] Redmill, F., M. Chudleigh, and J. Catmur, "System Safety: HAZOP and Software HAZOP," John Wiley and Sons Ltd, 1999.

[31] RTCA, I., "RTCA/DO-178B: Software Considerations in Airborne Systems and Equipment Certification," (1992).

[32] Szyperski., C., "Component Software: Beyond Object-Oriented Programming," Addison-Wesley, 2002, 2nd edition.

[33] UK Ministry of Defence, "Def Stan 00-55: Requirements for Safety Related Software in Defence Equipment," .