Automatic architecture hardening using safety patterns

Kevin Delmas, Rémi Delmas, and Claire Pagetti

ONERA/DTIM, 2 av. E. Belin, 31055 Toulouse, France
firstname.lastname@onera.fr

Abstract. Safety critical systems or applications must satisfy safety requirements ensuring that catastrophic consequences of combined component failures are avoided or kept below a satisfying probability threshold. Therefore, designers must define a *hardened architecture* (or implementation) of each application, which fulfills the required level of safety by integrating redundancy and safety mechanisms. We propose a methodology which, given the nominal functional architecture, uses constraint solving to select automatically a subset of system components to update and appropriate safety patterns to apply to meet safety requirements. The proposed ideas are illustrated on an avionics flight controller case study.

1 Introduction

The design and development of safety critical applications must satisfy stringent dependability requirements. Certification authorities even request the correctness of embedded avionic applications to be proved using formal arguments.

Context To help designers reaching that objective, several avionics standards are available. The aerospace recommended practices (ARP) 4754 [21] is a guideline for development processes under certification, with an emphasis on safety issues. Any avionics function is categorized according to the severity of its loss and subject to qualitative and quantitative safety requirements. For instance, a function categorized as CAT (catastrophic) shall not fail with strictly less than three basic failures and the probability of loss shall not exceed 10^{-9} per flight hour. High-level functions are then refined to a *preliminary functional architecture*, that is, their *implementation* is designed as a combination of sub-functions providing the expected functionality. This preliminary architecture is then analyzed to check if high level safety requirements are fulfilled assuming some properties (such as failure independence, failure modes and propagation rules). This design activity is iterative : in case the functional architecture does not satisfy the safety requirements, the designers must propose a new architecture with additional redundancies. We intervene at that level by automating the choice of *safety patterns* to use in order to tolerate hardware failures.

Contribution Our work consists in helping the designer to define a *hardened architecture* (or implementation) of each application fulfilling the required safety requirements by integrating redundancy and safety mechanisms. The hardening mechanisms to be used are often chosen depending on the designer's expertise and by considering the impact on the design of non-functional criteria (such as CPU consumption, temporal performance, etc). Usually, the experts rely on well-formed safety design patterns of proven efficiency [12,20]. To determine which design pattern to use, experts usually rely on guidelines or decision trees [2,19]. In this paper we propose a methodology to automate the hardening process with respect to qualitative safety requirements, whereby the minimum number of basic failure events required to trigger a failure condition shall be above a a certain threshold. The main steps of the proposed design process are: 1) Safety assessment: the functional application

architecture without any safety mechanism (nominal architecture) is first analyzed with regards to qualitative safety requirements. We use the AltaRica language [3] to model the architecture's dysfunctional behaviour in a modular way. We use the graphical IDE Cecilia OCAS [6] developed by Dassault to generate all the failure sequences that lead to a failure condition. A sequence is a list of events of AltaRica components leading the system to a given error state. 2) Hardening strategy: once the minimal sequence set is generated, we generate a pseudo-Boolean optimization problem is generated and solved automatically. The solution indicates a subset of system components that invalidate the safety requirements (involved in a minimal event sequence the size of which is too small) and, for each component, indicates the safety pattern to apply in order to increase the size of the invalid event sequence. 3) Component Substitution: Each selected component is then replaced by its hardened version in the system. The new system has a necessarily improved minimal sequence set.

This *Assessment, Hardening, Substitution* process is repeated and the system is modified until an architecture satisfying the safety requirements is obtained. In practice very few iterations are needed, since all non-satisfying minimal event sequences are dealt with and improved in each iteration.

In this work, we consider high level fault-tolerance mechanisms based on spatial or temporal redundancy, which can be implemented in several ways on many-core target architecture. The actual implementation of safety patterns on the many-core chip is future work and not addressed in this paper.

Related work Automatic architectural optimization of fault tolerant systems, under qualitative constraints and with respect to multiple quantitative criteria has been addressed in several previous papers [18], [1], [8], [24]. Most of these earlier approaches use genetic algorithms to explore the search space. Genetic algorithms breed numerous alternative evolutions of an initial architectural design while continuously assessing their fitness according to qualitative fault-tolerance properties and various other non-functional aspects encoded as criteria. They attempt to enumerate the pareto-front of the solution space to propose all possible design trade-offs to the user.

The work presented in this paper pursues identical goals, however the modelling formalisms and most importantly the design optimization techniques differ. First, we use the Altarica language and tools to specify system models and perform the qualitative safety evaluation. HiP-HOPs could have been a viable option, but it is more anchored in the automotive domain, while Altarica is well established in the aerospace domain and enjoys very mature qualitative safety evaluation tools. Altarica however contains no specific means of specifying the design space to be explored for architecture optimization. As a consequence, the notions of *design pattern*, *pattern instantiation* and *component-pattern substitution* need to be handled externally. Modelling frameworks such as EAST-ADL, designed with product-line specification in mind, offers a built-in *implementation variability* notion, and our method could fairly easily be adapted in such frameworks. The most salient feature of our approach is the way in which the optimization problem is formalized and solved: In a quick preprocessing phase, we generate a local failure model for each possible safety pattern, and from them we derive an ordering of safety patterns representing the fault-tolerance increase they offer, in all cases. This pattern ordering is then embedded in the optimization problem given to the constraint solver. This preprocessing and embedding techniques allows to totally avoid the typical computational bottleneck of most (if not all) genetic approaches to design optimization: the thorough qualitative evaluation (minimal sequence or minimal cutset generation) of the numerous candidate solutions in each generation. Our method offers far greater efficiency and scalability.

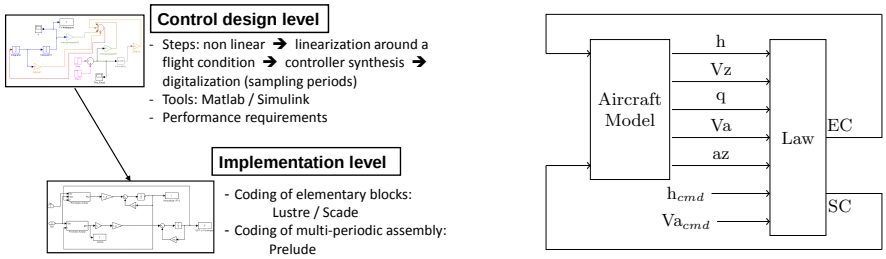
The rest of the paper is organized as follows: section 2 presents the case study. Section 3 explains the general ideas of the proposed hardening approach. Section 4 details how to generate the pseudo-Boolean optimization instance which, when solved, yields a hardening strategy for the

system and provides results obtained on the case study. Last, section 5 concludes the paper and outlines perspectives to this work.

2 Case study

In this section, we present a control-command application that is hosted on a muli/many-core platform and has to be hardened against hardware failures of the execution platform.

The ROSACE case study We consider the open-source avionics control engineering case study ROSACE defined in [17], managing the longitudinal motion of a medium-range civil aircraft in *en-route* phase.



(a) Functional development cycle

(b) Functional architecture

Fig. 1. ROSACE overview

The figure 1(a) roughly depicts the functional development cycle for such a controller. The control engineers model the application with MATLAB/SIMULINK [23], analyze the model behaviour and synthesize controllers satisfying performance and robustness properties. The figure 1(b) details the functional interface between the controller (*Law*) and the *Aircraft model* that represents the system to be controlled (aircraft, engines and elevators). The controller must satisfy several properties, in particular control performance properties (*eg overshoot*, *settling time*, ...), which are validated within SIMULINK by intensive simulation.

The MATLAB/SIMULINK specification is then translated (manually or automatically) to (1) a set of SCADE/LUSTRE programs [7,9] and (2) a multi-periodic assembly expressed in a home made language or in PRELUDE [16], as proposed in the original ROSACE paper. The functional correctness of the implementation is validated via intensive simulation, compliant with the activities at the MATLAB/SIMULINK level, to verify the performance properties. The controller can then be ported on a real target.

Challenges for embedding multi/many-core COTS processors Multi/many-core processors will inevitably become the only COTS available for the development of embedded applications. However these platforms, originally designed for performance, suffer several drawbacks for safety critical application contexts:

- due to their internal complexity (caches, buses, branch prediction units, etc), deterministic execution and safe evaluation of worst case execution times (WCET) of software functions is almost impossible [26];
- they are subject to more kinds and more frequent hardware faults: the high density integration of many small devices on a single chip increases the occurrence *permanent failures* due to various phenomena such as aging, wear-out or infant mortality [4]. They also encounter more *intermittent faults*, caused by manufacturing, thermal or voltage variations, that can generate regular bursts that may last several cycles [25]. They are also confronted to *transient faults* and in particular those resulting from *Single Event Upsets* (SEU) [11].

So, with this paper we promote the idea that if many-core chips are to be used in critical contexts, safety requirements shall also be an integral part of their design process. In this paper we mainly consider faults occurring on multi/many-core COTS processors executing a safety critical control application. This means that we assume that any memory region such as registers, caches or buffers in the NoC (network on chip) can be corrupted by an SEU, in the sense that any cell's content can be modified in an unforeseen way because some of its bits was flipped. Usually, the DDR-RAM is protected against those faults [15]. An SEU can affect: (1) a data, if it is never used, the fault remains dormant but if the erroneous value is used, this could lead to an incorrect result or run-time error (*eg* division by zero); (2) an operation (or OP-CODE); (3) the control flow (jumps). So, even though corruption events are more likely to occur on multi/many-core platforms, the huge quantity of cores and resources they offer (memory, NoC bandwidth) makes them nonetheless worth considering for critical applications: firstly, software patterns originally designed to tolerate SEUs on uni-processor systems [5,22] can easily can be extended to multi/many-core platforms, secondly and most importantly, new fault-tolerance strategies, based on extensive software and communication redundancy, imparting resource consumption levels once considered prohibitive for uni-processor systems, seem very interesting and must be explored.

Fault model The original implementation of ROSACE is nominal in the sense that no special fault-tolerance mechanisms is included to tolerate failures of the execution hardware. A first usual hardening is made at the functional level, *ie* in the MATLAB/SIMULINK specification. The purpose is to support external failures, of the sensors for instance. The specification is modified using standard methods such as analytic redundancy (exploiting mathematical invariants of the controller) or Kalman filters [14]. We have modified the MATLAB/SIMULINK specification in that way with the help of David Saussié. In the sequel, we will use a version of the *Law* function that can tolerate the failure of either sensor V_a, V_z, h, a_z or q, in erroneous or loss mode, without being degraded. Indeed, thanks to analytic redundancy techniques, any missing or erroneous value of V_a, V_z, h, a_z or q can be reconstructed from the others. The details of these modifications are out of the scope of this paper.

We then want to generate a *hardened* version of the controller to support SEU-induced failures of the execution platform. We first detail our fault model, *ie* the possible effects of SEUs on the application. We will consider that *EC*, *SC* are out of the scope of safety analysis. We applied a FMECA (Failure mode, effects and criticality analysis) to determine them:

Element	Failure modes	explanation
h, V_a, V_z, q, a_z	Temporary erroneous (e^t)	involved in computation in Law
	Temporary erroneous (e^t)	provide erroneous orders on the actuators
Law	Permanent loss (l^p)	run-time error
	Temporary loss (l^t)	OP-CODE error
	Temporary erroneous (e^t)	false result
	Permanent erroneous (e^p)	faults in integrator (*eg* when h.e^t)

Both the erroneous behavior and total loss of the controller are classified as CAT (catastrophic), therefore the safety requirements on the execution on the multi/many-core platform are:

P1 No single or double error shall lead to the loss of the control function
P2 No single, double or triple error shall lead to the erroneous control of the actuators

The minimal event sequences leading to the failure conditions for the nominal design are given below:

Failure condition	Event sets (ordering irrelevant)	
	of size 1	of size 2
loss of the control	$\{Law.l^p\}$	$\{S_1.l^p, S_2.l^p\}$ where $S_1, S_2 \in \{V_a, V_z, h, a_z, q\}$ and $S_1 \neq S_2$
erroneous control	$\{Law.e^p\}$	$\{S_1.e^p, S_2.e^p\}$; $\{S_1.l^p, S_2.e^p\}$ where $S_1, S_2 \in \{V_a, V_z, h, a_z, q\}$ and $S_1 \neq S_2$

3 Overview of the Proposed Hardening Approach

Given an initial system architecture, we address the problem of automatically increasing its fault-tolerance level, while preserving its functionality. To do so, we propose to use safety design patterns and combinatorial optimization under pseudo-Boolean constraints. This section introduces notations used in the rest of the paper, and a synthetic overview of the proposed method.

3.1 Design Methodology Overview

The main steps of the proposed design process are depicted in the figure below.

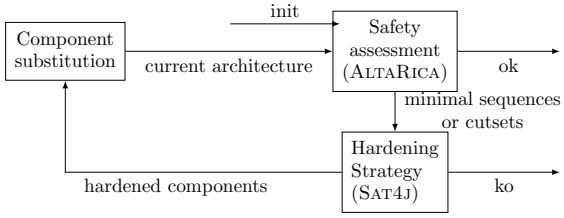
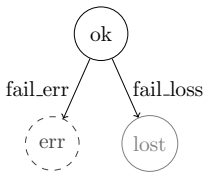


Fig. 2. Hardening Process

Safety Evaluation The initial architecture is first analyzed with regards to qualitative safety requirements. We use the AltaRica modelling language [3] to represent the architecture's dysfunctional behaviour. AltaRica is based on extended finite automata which can exchange values of specific variables (named data-flow variables) and which can be synchronized (synchronized product or broadcast). The idea is to describe the failure modes of a component as different states of an automaton. Transitions between states are triggered by the occurrence of failure events, and data-flow equations are used to define the propagation of failure modes through components, from input to output depending on their internal state. As an example, let us consider a simple *voter* function, with three input flows (i_1, i_2, i_3) and a single output flow o, represented in the figure below. The automaton on the left represents the possible internal failure states of the voter, while equations on the right describe the error propagation behaviour from input to output, depending on the currently active state.



$$\text{case ok: } o = \begin{cases} ok \ \textit{if} \ i_1, i_2, i_3 = \{ok, ok, \star\} \\ lost \ \textit{if} \ i_1, i_2, i_3 = \{ok, lost, lost\} \\ err \ \textit{otherwise} \end{cases}$$

$$\text{case lost: } o = lost$$

$$\text{case err: } o = err$$

The possible states of the voter are: the nominal mode *ok* and the failures modes *lost* and *err*. Initially, the voter is *ok*. If one of the *err_fail* or *err_lost* failure event occurs, the corresponding failure state is entered. On the right hand side, the data-flow behaviour is schematized. The voter takes as input three flows (i_1, i_2, i_3) and forwards a consolidated flow o as output. The output failure status depends both on the failure status of the inputs and on the voter's own failure mode. If the voter is in its nominal mode, the output is: *ok* as long as at least two inputs are *ok*; *lost* if two input flows are *lost*; *err* if at least two inputs happen to have an erroneous status. If however the voter itself is *lost*, the output is *lost* as well; if the voter is in the *err* state, the output is *err* as well.

So, AltaRica components are an abstraction of the entity they represent, with an internal state, basic events trigger internal state changes, and to each state corresponds a distinct set of dataflow equations linking inputs and outputs (in that sense the outputs of a component are always directly dependent on its current inputs and current internal state). The dataflow types in the AltaRica model represent the failure statuses of dataflows, and not their actual numerical value. For instance, if an ADIRS component generates an output o of type "3D vector of floating-point", its AltaRica abstraction will have a corresponding dataflow o with an enumerated type $enum\{ok, err, lost\}$, representing the fact that the 3D floating-point vector is either correctly produced, erroneously produced or not produced at all by the component. In such models, it is often the case that a component produces erroneous outputs when given erroneous inputs, even if not in a failed state: it is simply the result of a propagation and is not considered as a failure. One other important remark is that the semantics of the AltaRica language is such that state transitions in a component are only triggered by basic failure events and never by immediate combinations of input flows. Unless specified otherwise, all basic events are considered independent (transitions can however be guarded by conditions on flows).

Modelling in AltaRica is done using the graphical IDE Cecilia OCAS [6] developed by Dassault. It allows to define component libraries: this is particularly interesting to capitalize on safety patterns. The framework also offers the possibility to generate the set of minimal failure sequences that lead to a failure condition, or the set of minimal cutsets, when the order of occurrence of events is irrelevant in the model.

A *minimal sequence* is a list (order significant) $[e_1, \ldots, e_n]$ of basic failure events of AltaRica components leading the system to a given error state. When the model is *static*, that is, when the occurrence of any given set of events always leads the system to a same state, regardless of their order of occurrence, a *minimal cutset* $\{e_1, \ldots, e_n\}$ denotes a minimal set of basic failure events leading the system to a given error state.

In particular a dysfunctional model of the initial Rosace Simulink specification was built in AltaRica. The minimal failure sequences obtained for this model are given in the table found at the end of section 2.

Hardening strategy Once the minimal event sequences have been generated, either their cardinality either satisfy the safety requirements and the current architecture is good enough, or else we need to apply more safety patterns on some well chosen components. The hardening decisions are performed automatically in two steps: (1) **Component Selection**: a subset of components of the system is selected for pattern application based on the events present in the sequence needing a size improvement. (2) **Pattern Selection**: for each selected component, a single hardening pattern is selected from a library of applicable patterns. Pattern composition is not allowed, which makes the analysis bounded yet possibly sub-optimal. However, in practice this restriction is not harmful.

As will be seen in section 4, the main novelty and strength of the proposed process with respect to traditional approaches is the simultaneous and automatic resolution of both the *component selection* and *pattern selection* problems using a combinatorial optimization technique: based on the system's minimal sequence set (or set of minimal cutsets) and on an ordering of safety patterns obtained through a preprocessing step, a pseudo-Boolean optimization problem is generated and solved using the state-of-the-art solver (for instance SAT4J [13], or any similar solver).

Component Substitution Each selected component is then replaced by its hardened version in the system. This operation increases the size of at least one minimal event sequence in the set of unsatisfactory sequences, while not reducing the size of others. A new iteration is started on

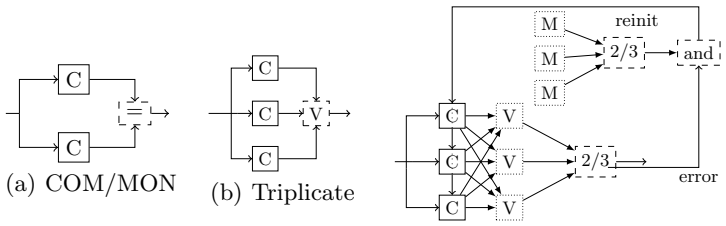
the modified system, beginning with a qualitative safety evaluation pass to determine if further improvement is needed.

3.2 Considered safety patterns

In this paper, we view *components* as types which define input and output interfaces and behaviour which can be *instantiated* and interconnected to yield an analyzable system. For instance a component ADIRS (Air Data Inertial Reference System) can have several instances ADIRS_A, ADIRS_B, etc. in a system. A safety design pattern is, however, an abstract design with known fault-tolerance properties, expressed in terms of one or more *component parameters* (place-holders). Such parameterized designs are viewed as parameterized types $P\langle C\rangle$, where $\langle C\rangle$ denotes the parameter. The parameter C of a pattern must be *instantiated* with an actual component type to yield a new, non-parameterized component type which can, in turn, be instantiated to yield a concrete instance usable in a system. For instance, the pattern COM/MON$\langle C\rangle$ is instantiated with the ADIRS component to yield a component type "COM/MON of ADIRS" COM/MON_$ADIRS$, which can itself be instantiated one or several times in a given system. Many patterns exist in the literature. Fig. 3 illustrates three of them. The graphical conventions are as follow: dashed rectangles represent safety mechanisms assumed to be fail-free and dotted rectangles represent safety mechanisms that can fail. Obviously, the pattern parameter C can fail. Fig. 3(a) is a COM/MON (command and monitoring) pattern. Two redundant components work in hot redundancy, computing a same value. A comparator (assumed perfect) checks if the outputs are coherent, if so the consolidated value is forwarded, otherwise no output is provided. This pattern does not increase the minimal event sequence size for failure conditions of the *loss* type (if one C is lost the comparator blocks the output) but transforms an *erroneous* into a *loss*. Fig. 3(b) is a triplication where three identical components compute in parallel the same value and a voter (assumed perfect) elaborates a correct value from them. Fig. 3(c) is more complex: it is based on triple modular redundancy where the C components have permanent failure modes repairable by hot reinitialization, M are memory components which errors are periodically repaired, and V are memory-less median voters with transient failure modes (the whole voter state is refreshed at every execution of the function). Each of the three V receives a numerical value from each C, produces a consolidated output value and diagnoses which of the three C is currently failing (deviation from the median above a set threshold). The function '2/3' is a classic two thirds majority vote, assumed fail-free, which consolidates the V outputs, *ie* the pair *(output_value, failing_C)*. A single permanent C-error will be diagnosed coherently by all three V and pass the 2/3 vote. However, a single transient V-error will be filtered out by the 2/3 vote, so no false positive or negative C-error detection is possible. The state of each C is periodically saved to its corresponding M, as long as no C-error is diagnosed. A C reinitialization is triggered when all three V identify a same failing C, and is achieved by replacing the failed C's internal state with a safe state elaborated from the M values using the 2/3 voting (so that any single permanent M-error is filtered out), and by resuming the component's operation in the *ok* state. Under the assumption that a C-error is always repaired before the next independent C-error, and that M-errors are periodically fixed by saving a valid C state in each M, this pattern has a better SEU fault-tolerance than a simple triplex.

We will mostly use the patterns based on replication in Fig. 3(b) and Fig. 3(c) since we need to detect and tolerate *erroneous* behaviours. When a component is replicated in a pattern, each instance is considered failure-independent from the others.

Let us consider a component C with two failure modes *lost* and *err*. We detail below the minimal event sequences obtained when applying each of the patterns proposed above. We denote by C^i the i-th instance of C in the pattern:



(a) COM/MON (b) Triplicate (c) Triplicate and repair

Fig. 3. Patterns

Pattern	Failure mode	Basic event sets (or sequences)
Fig. 3(a)	$P\langle C\rangle.lost$	$\{\{C^1.lost\},\{C^2.lost\},\{C^1.err\},\{C^2.err\}\}$
	$P\langle C\rangle.err$	$\emptyset$
Fig. 3(b)	$P\langle C\rangle.lost$	$\{\{C^i.lost, C^j.lost\},\{C^i.err, C^j.lost\}\}$ with $i \neq j$
	$P\langle C\rangle.err$	$\{\{C^i.err, C^j.err\}\}$ with $i \neq j$
Fig. 3(c)	$P\langle C\rangle.lost$	$[M^i.lost, M^j.lost, V^1.l^t, V^2.l^t]; [M^i.lost, M^j.lost, C^k.l^p, C^l.l^p];$ $[M^i.lost, M^j.lost, C^k.l^p, C^l.e^p]$
	$P\langle C\rangle.err$	$[M^i.err, M^j.err, V^1.e^t, V^2.e^t]; [M^i.err, M^j.err, C^k.e^p, C^l.e^p]$

The patterns used in our approach all use a combination of component replications and functionally neutral failure detection and recovery mechanisms (voter, comparator, switch), which preserve the component data-flow interfaces. We will assume that these are sufficient arguments to allow us to say that pattern application, which consists in replacing a component C by $P\langle C\rangle$, preserves the original functionality.

4 Automatic Hardening

After defining the notion of pseudo-Boolean constraint system, we detail notations and how to formalize the component and pattern selection problems as pseudo-Boolean constraints and optimization criteria. The ROSACE application used for illustration is a dynamic system so minimal event sequences are used to explain and illustrate the method, however the method works in the very same way for static models for which minimal cutsets are used.

4.1 Pseudo-Boolean Constraint Systems

A pseudo-Boolean constraint system expresses constraints over a set of pseudo-Boolean decision variables $\{x_i\}_i$, *ie* boolean variables interpreted as $O/1$ integers. A *literal* l_i is a pseudo-Boolean variable x_i (positive literal) or its logical negation $\neg x_i$ (negative literal). Negative literals can be eliminated in favor of positive literals by rewriting $\neg x$ to the equivalent $(1 - x)$. A *linear pseudo-Boolean constraint* has the following form: $[a_1.l_1 + \cdots + a_n.l_n \ \langle \text{rel} \rangle \ k$. Where the dot '.' represents scalar multiplication, $a_1, \ldots, a_n \in \mathbb{Z}$ are relative integers (and can be omitted when equal to 1), $l_1, \ldots, l_n$ are decision literals, with underlying variables $x_1, \ldots x_n$ and no repetition, $\langle \text{rel} \rangle \in \{<, \leq, =, \geq, >\}$, $k \in \mathbb{N}$ is a positive integer.

An *interpretation* I for a pseudo-Boolean constraint system is a total mapping $I : \{x_i\} \mapsto \{0, 1\}$. To evaluate a constraint C according to an interpretation I, we proceed first by substituting each l_i appearing in its left hand side with $I(x_i)$ if $l_i = x_i$, or with $(1 - I(x_i))$ if $l_i = \neg x_i$; second, by evaluating and summing the monomials, yielding a integer value v and by comparing v with the constant k at the right hand side according to the specified relation $\langle \text{rel} \rangle$. If the relation is satisfied, the constraint is said to be *satisfied*, otherwise it is said to be *violated*. An interpretation I is called

a *model* of the constraint system if and only if each constraint is satisfied under I. In the remaining we will denote models with the letter M instead of I.

Example 1. Assuming $I = \{x_1 \mapsto 0, x_2 \mapsto 1, x_3 \mapsto 1, x_4 \mapsto 0\}$, the constraint $2.x_1 + 3.\neg x_2 + 10.x_3 + 1.\neg x_4 > 10$ is satisfied, while the constraint $1.\neg x_1 + 2.\neg x_2 + 1.x_3 + 1.\neg x_4 = 4$ is violated.

A model is qualitatively assessed by defining one or more numerical criteria over it. An *optimization criterion* is any linear pseudo-Boolean expression over the problem variables, of the form: $K \equiv a_1.l_1 + \cdots + a_n.l_n$, where: $a_i \in \mathbb{Z}$ and l_i are literals.

A criterion is evaluated under an interpretation just as left hand sides of constraints are. Given a single criterion and a constraint set we define the notion *min-optimality*: M is *min-optimal* with respect to the criterion if and only if there exists no other model yielding a smaller value for the criterion. Given a finite criteria list[1] $[K_j]$, a set of constraints $\{C_i\}$ we define the notion of *leximin-optimality* for a model M: M is *leximin-optimal* with respect to $[K_j]$ if and only if M is min-optimal for K_1 and, for all $j > 1$, M is min-optimal for K_j on the extended constraint set $\{C_i\} \cup \{K_{l-1} = M(K_{l-1}) | l \in [1, j]\}$, where $M(K_{l-1})$ is the min-optimal value taken by K_{l-1} in M.

4.2 System and Safety Pattern Characterization

System Model and Notations. A candidate system is characterized by the following sets and functions:

- MSS (respectively, MCS): function which generates the minimal sequences set for a system (respectively, the set of minimal cutsets for a static system);
- Comps: the set of system components;
- Events: the set of all basic failure events that can be fired in the system;
- Comp2Evt : Comps $\mapsto 2^{\text{Events}}$ returns for each component the subset of events belonging to that component (the set of events admits a partition indexed by the set of components);
- Evt2Comp : Events $\mapsto$ Comps returns for each event the component to which this event belongs.

Remark: In order to generate the pseudo-Boolean constraint system to solve, given a minimal sequence of events $[e_1, ..., e_n]$ we only need to consider the underlying set of events of the sequence, noted $e_1, ..., e_n$ where the ordering of elements is irrelevant. So in the rest of this section, we always view minimal sequences (and obviously minimal cutsets) as sets of events.

Safety Pattern Model and Notations. Safety design patterns are abstract designs, parameterized by one or more *component parameters* with known fault-tolerance properties. Patterns are noted $P\langle C\rangle$, where $\langle C\rangle$ denotes the parameter. Parameters must be instantiated with actual components to yield a concrete design. The set of all patterns, in their abstract, parameterized form is noted Pat.

Example 2. Consider for instance the *triplication and voting* pattern of Fig. 3(b): Triplicate$\langle C\rangle \equiv$ Vote(C^1, C^2, C^3). It takes a single component C as parameter and yields a new design in which C is triplicated into C^1, C^2 and C^3 (superscripts are used to distinguish multiple instances of a same component (assumed to fail independently) which outputs are then consolidated using a triplex voter (using for instance median, majority or any other pertinent voting method). This pattern provides a fault-tolerance improvement, since the output is now tolerant to one fault of either C^1, C^2 or C^3, and allows to detect two faults, whereas the original component C tolerates zero faults (the formal criterion used to assess the improvement provided by a safety pattern is detailed later in section 4.2).

[1] order is significant

The interface of a component C consists of a list of typed input data-flow variables and a list of typed output data-flows:

$$\text{InFlows}(C) = [o_i : t_i | i \in [1, N_I]]$$
$$\text{OutFlows}(C) = [o_i : t_i | i \in [1, N_o]]$$

Where N_i, N_o are the numbers of input and output flows of the component, each t_i is an enumerated type of the form $\texttt{enum}\{\texttt{ok}, \texttt{fault}_1, \ldots, \texttt{fault}_n\}$, in which the value `ok` is always present and represents the "no fault" state (imposed by design guidelines).

To be usable in our approach, in addition to providing the same functionality as the parameter component, a design pattern needs to preserve the output interface of the parameter component :

$$\text{OutFlows}(P\langle C\rangle) = \text{OutFlows}(C) \tag{1}$$

Since design patterns may have some restrictions which make them only applicable to certain components, we assume that for each component C the following function provides us with the subset of safety patterns applicable to C:

$$\begin{aligned}\text{CompPat} : \text{Comps} &\mapsto 2^{\text{Pat}} \\ C &\mapsto \{\text{Id}\} \cup \{P_1, \ldots, P_n\}\end{aligned}$$

where the distinguished element Id, the *identity pattern*, equal to the component itself without modification, is always available.

An Order Relation Over Safety Patterns. We now need to define a formal characterization of the effect, on the minimum size of minimal event sequences of a system, of applying a design pattern to a component. Intuitively, replacing a component C by $P\langle C\rangle$ should augment the number of failure events needed to reach a same faulty state, or even render that state unreachable, which should entail an improvement of the minimal event sequence set of the system hosting this hardened component, either by increasing the size of some sequences, or by removing sequences.

Due to the modeling guidelines used in the proposed approach, the fact that a component C enters a faulty state after firing an event e necessarily manifests itself at the output interface of the component, in which at least one the output flows takes a value different from the `ok` value. So, a component is in a faulty state whenever the following predicate over the component outputs evaluates to $\top$:

$$\text{Faulty}(C) \equiv \bigvee_{o \in \text{OutFlows}(C)} o \neq ok \tag{2}$$

In order to determine the general increase of fault tolerance level we can obtain from using a pattern $P \in \text{CompPat}(C)$ instead of C in the system, we will compute all combinations of events allowing to reach, in $P\langle C\rangle$, any fault state observationally equivalent[2] to some fault state of C with respect to the formula defined in (2).

Thanks to the output interface preservation property (cf. Equation (1)), it is possible to compare C and $P\langle C\rangle$ through the formula defined in (2).

More precisely we can generate the minimal event sequences (or minimal cutsets) for the failure condition $\text{Faulty}(P\langle C\rangle)$ on the component $P\langle C\rangle$, in isolation outside of the context of the system[3]

[2] two components with matching output interfaces are in observationally equivalent states with respect to some formula expressed over their outputs flows if the formula evaluates to true for both components

[3] The minimal sequence set generation tool of the AltaRica tool suite easily allows us to obtain the desired result.

under the assumption that its inputs have an *ok* status, in order to determine the influence of the pattern's own failure events on the statuses of its outputs. Such sets of minimal sequences (or minimal cutsets MCS) are noted as follows:

$$\text{MSS}(\underbrace{P\langle C\rangle}_{\text{component}}, \underbrace{\text{Faulty}(P\langle C\rangle)}_{\text{failure condition}})$$

The case $\text{MSS}(P\langle C\rangle, \text{Faulty}(P\langle C\rangle) = \emptyset$ could mean: no basic event is needed to reach failure condition; or the failure condition is not reachable. As we assume that any system must be in a functioning state in nominal mode, an empty minimal sequences set means that current architecture is acceptable (not fallible) and the hardening process stops. Once we have the set of minimal sequences (or set of minimal cutsets) for both C and $P\langle C\rangle$, we need to define some criterion that will allow us to soundly measure the increase of fault tolerance provided by the pattern. To do so we propose to compare the minimum cardinality of sequences (or cutsets) present in each set. So, assuming a function card which returns the cardinality of a set or sequence of events, we first define the function which returns the size of the smallest event set in a set of event sets:

$$\text{MinCard}(\text{MSS}) = min\{\text{card}(ms) | ms \in \text{MSS}\}$$

Last, if the pattern $P\langle C\rangle$ satisfies the following relation:

$$\text{MinCard}(P\langle C\rangle, \text{Faulty}(P\langle C\rangle)) > \text{MinCard}(C, \text{Faulty}(C)) \tag{3}$$

it indeed provides an increase of fault tolerance level over C *in any situation*, since it takes strictly more events to reach a fault state in $P\langle C\rangle$ than it takes to reach any observationally equivalent fault state in C.

We can generalize this idea and compare not only a pattern against a base component but also a pattern against a pattern, by defining an order relation $>_{\text{CompPat}(C)}$ over the set of patterns $\text{CompPat}(C)$:

$$P >_{\text{CompPat}(C)} P' \equiv \text{MinCard}(P\langle C\rangle, \text{Faulty}(P\langle C\rangle)) > \text{MinCard}(P'\langle C\rangle, \text{Faulty}(P'\langle C\rangle)) \tag{4}$$

In our approach, this order relation must be computed *a priori* for each component, by instantiating each pattern available for this component and generating minimal sequences for the Faulty failure condition (cf. Equation (2)).

Example 3. Let us consider a component Sensor with two failure events Sensor.*lost* and Sensor.*err* and the *Triplicate* pattern introduced earlier. Assuming that the Vote component itself has no failure mode, we get the following results, compliant with previous results given in section 3.2:

	Sensor	$\text{Triplicate}\langle\text{Sensor}\rangle \equiv \text{Vote}(\text{Sensor}^1, \text{Sensor}^2, \text{Sensor}^3)$
MSS	{Sensor.*err*}	$\text{Triplicate}\langle\text{Sensor}\rangle.err = \{\text{Sensor}^i.err, \text{Sensor}^j.err\}$ *with* $i \neq j$
	{Sensor.*lost*}	$\text{Triplicate}\langle\text{Sensor}\rangle.lost = \begin{cases}\{\text{Sensor}^i.err, \text{Sensor}^j.lost\} \\ \{\text{Sensor}^i.lost, \text{Sensor}^j.lost\}\end{cases}$ *with* $i \neq j$
MinCard(MSS)	1	2

So in conclusion $Triplicate\langle Sensor\rangle >_{\text{CompPat}(Sensor)} Sensor$.

Component Substitution Model. Last, we need to trace the system's evolution from one hardening iteration to the next. For that we define the function:

$$\text{PrevPattern} : C \in \text{Comps} \mapsto P \in \text{CompPat}(C) \tag{5}$$

which returns for each component the pattern previously applied to that component, or the identity pattern if none was applied.

4.3 Component and Pattern Selection Constraints Generation

To model the component selection problem, we begin by introducing new a set of pseudo-Boolean variables.

Definition 1 (Component Selection Variables). *The set of* component selection variables *is defined as:* $\{\text{SelectComp}(C)|C \in \text{Comps}\}$*, where* $\text{SelectComp}(C) = \top$ *if and only if component* C *is selected for pattern application.*

We then consider a set MSS of minimal sequences, such that each element has a cardinality below the threshold required by the classification of some failure condition. We must make sure that at least one component involved in each minimal sequence will be selected for pattern application. We do so by generating a component selection constraint for each element of the set MSS using the following definition.

Definition 2 (Component Selection Constraint). *The component selection constraint for a minimal sequence* $ms = \{e_1, \ldots, e_n\}$ *is defined as:* $\text{SelectCompCtr}(ms) \equiv \sum_{e \in ms}(\text{SelectComp}(\text{Evt2Comp}(e))) \geq 1$.

The problem of selecting a minimal set of components allowing to cover a given set of minimal sequences (more exactly, cutsets) was previously addressed in [10], using a prime implicant calculus for formulas in disjunctive normal form. In the approach proposed here however, we use pseudo-Boolean constraints to achieve the same goal, because it allows us to integrate and solve the pattern selection constraints in a unified framework. Consequently, to model the pattern selection problem, we begin by introducing new a set of pseudo-Boolean variables.

Definition 3 (Pattern Selection Variables). *The set of* pattern selection variables *for a component* $C \in \text{Comps}$ *is defined as:*
$\{\text{SelectPat}(C, P)|C \in \text{Comps}, P \in \text{CompPat}(C)\}$*, where* $\text{SelectPat}(C, P) = \top$ *if and only if pattern* P *must be applied to component* C*.*

For each component C, the following constraint is instantiated, to make that at most one applicable pattern is selected: $\text{AtMostOnePatternCtr}(C) \equiv \sum_{P \in \text{CompPat}(C)} \text{SelectPat}(C, P) \leq 1$.

Next, in order to ensure that a pattern gets selected for a component *if and only if the component itself is selected*, the following constraint is instantiated for each component $C \in \text{Comps}$ and each pattern $P \in \text{CompPat}(C)$:

$$\text{SelectCoherenceCtr}(C, P) \equiv \neg\text{SelectComp}(C) + \text{SelectPat}(C, P) \leq 1$$

Furthermore, the pattern selected for a component must be better, according to the order relation $>_{\text{CompPat}(C)}$ defined in Equation (4), than the previously selected pattern for that component, which remember is given through the function PrevPattern defined in (5). However, in order to be able to express this constraint at pseudo-Boolean level, we need define the embedding of the pattern ordering relation in the pseudo-Boolean constraint system, which needs us to introduce specific pseudo-Boolean variables and constraints.

We assume that the pattern ordering relation is defined for each component C and each pair of applicable patterns $(P, P') \in \mathit{componentPatternSet}(C)^2$ through the following set of variables:

$$\{\text{GT}_C(P, P')|C \in \text{Comps}, (P, P') \in \text{CompPat}(C)^2\}$$

With the semantics that $\text{GT}_C(P, P') = \top$ if and only if $P\langle C\rangle >_{\text{CompPat}(C)} P'\langle C\rangle$.

The relation itself, *ie* the truth-values for these variables, can be encoded using a set of constraints of the form:

$$\text{OrderRelationCtr}(C, P, P') \equiv \begin{cases} \text{GT}_C(P, P') > 0 \text{ if } P\langle C\rangle >_{\text{CompPat}(C)} P'\langle C\rangle \\ \neg\text{GT}_C(P, P') > 0 \text{ if } P\langle C\rangle \leq_{\text{CompPat}(C)} P'\langle C\rangle \end{cases}$$

Last, we instantiate the following constraint for each $C \in$ Comps and each pattern $P \in$ CompPat(C):

$$\text{BetterThanPrevious}(C, P) \equiv \neg\text{SelectPat}(C, P) + \text{GT}_{\text{CompPat}(C)}(P, \text{PrevPattern}(C)) \leq 1$$

in order to make sure that the selected pattern for C, if any, is better than the previous one according to the pattern ordering relation (the previous pattern chosen for C is available through the PrevPattern function).

4.4 Optimization Criteria

As explained in section 4.1, when solving a constraint system it is possible to define one or more optimization criteria to guide the search towards solutions minimizing or maximizing some numerical quantitative ranking function(s).

A first useful optimization could be to minimize the number of components selected for pattern instantiation: NofComponents $\equiv \sum_{C \in componentSet}$ SelectComp(C)

Second, a variety of cost metrics can be associated with design patterns, for instance: CPU consumption, Memory consumption, cost of implementation of the design pattern, etc. These criteria will influence the selection of the components and of the patterns, but in all cases all constraints expressed in the previous section will be satisfied.

So, we can use leximin or pareto optimization to optimize a list of criteria reflecting the user's preferences: first minimize the number of modified components, then CPU cost, then memory cost, ... So the final optimization problem would look like this: *Minimize NofComponents, CPU Cost, ... Subject to:* $\langle$*component selection constraints*$\rangle$ *and* $\langle$*pattern selection constraints*$\rangle$.

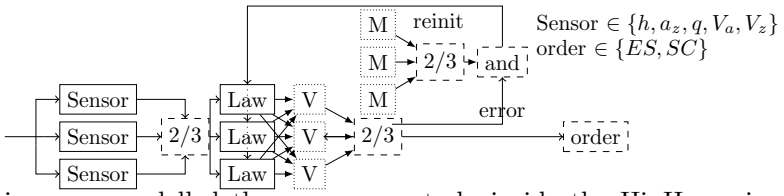
It is even possible for the user to define his own preference order over the set of patterns for each component, so that when two patterns are equivalent with respect to the $<_{\text{CompPat}(C)}$ ordering, the best according to the user preference gets selected.

4.5 Results

To sum up, the main steps in order to apply the proposed method:

1. Initialization step:
   (a) Define, for each component $C \in$ Comps the set of applicable patterns CompPat(C).
   (b) Instantiate each $P\langle C\rangle$ from that set, and compute minimal sequences for the Faulty($P\langle C\rangle$) failure condition.
   (c) Compute the order relation over CompPat(C) based on the MinCard criterion (cf. Eq. 3)
2. In each iteration:
   (a) Generate and filter out the minimal sequences that are too short for the required failure conditions;
   (b) Generate and solve the component selection and pattern selection constraints based on the current state of the design, with desired optimization criteria;
   (c) Based on the solver output, update the system architecture, and update the PrevPattern function definition (since it will be needed at the next iteration).

The hardened architecture for the case study is given in figure below.



For comparison, we modelled the same case study inside the HipHops simulink plugin, by annotating the simulink functional architecture with fault behaviour information required by HipHops. After one hour of computation, the design optimization function of Hip-Hops had not yet completed the first iteration of the genetic optimization routine, which consists in producing, by random pattern instantiation, an initial random population (an unknown number) of candidate models, and in evaluating these candidates by generating their fault trees and extracting minimal cutsets from them. Other numerical metrics are evaluated by HipHops, such as system availability, whereas they are not in our method. However, once cutsets are known, evaluating these metrics has a negligible cost, so this difference do not suffice to explain the observer runtime behavior, which suggests that fault tree and minimal cutset generation are indeed the real time-eater. So, our initial intuition about the better computational efficiency of our new method seems confirmed by this experiment, yet more benchmarks need to be addressed to augment the confidence in this result.

5 Conclusion and Future Work

In this paper we presented an approach to automate the task of hardening a given candidate system which does not meet qualitative requirements. Starting from the minimal sequence set (or minimal cutsets) of the system for some failure condition and from a collection of safety design patterns applicable to system components, we proposed a detailed method to generate a pseudo-Boolean constraint system which, when solved, proposes a detailed strategy for hardening the system, consisting of a selection of components and of associated safety patterns. While still at the proof of concept stage, the proposed method is generic, and the conditions the patterns must satisfy for the method to apply were precisely identified. Our optimization method was benchmarked against another well-known design optimizer on a medium-sized and realistic case study (ROSACE) and produced a satisfying answer orders of magnitude faster that the concurrent optimizer.

Future work will first consist in refining the logical safety patterns characterization to be able to deal with a greater variety of design patterns. We also aim at properly handling the difference between design patterns appropriate for *erroneous* failure conditions and those appropriate for *lost* failure conditions. Our current patterns ranking criterion simply does not distinguish between these two cases. A possible solution could be to generate distinct minimal sequence sets (or minimal cutsets) for each possible failure mode of a component and corresponding pattern, not just for some failure mode, to be able to precisely characterize the effect of each pattern on each possible mode, and base pattern selection on these metrics.

We also recently started working on extending the approach with quantitative safety indicators such as reliability or failure intensity, our goal is to be able to optimize automatically a design with respect to both qualitative and quantitative safety indicators. Last, we will also evaluate the scalability of the approach on several larger systems.

References

1. M. Adachi, Y. Papadopoulos, S. Sharvia, D. Parker, and T. Tohdo. An approach to optimization of fault tolerant architectures using hip-hops. *Softw., Pract. Exper.*, 41(11):1303–1327, 2011.
2. A. Armoush. *Design patterns for safety-critical embedded systems.* PhD thesis, 2010.

3. A. Arnold, G. Point, A. Griffault, and A. Rauzy. The altarica formalism for describing concurrent systems. *Fundam. Inform.*, 40(2-3):109–124, 1999.
4. S. Borkar. Designing reliable systems from unreliable components: The challenges of transistor variability and degradation. *IEEE Micro*, 25(6):10–16, Nov. 2005.
5. S. Campagna and M. Violante. An hybrid architecture to detect transient faults in microprocessors: An experimental validation. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'12)*, pages 1433–1438, San Jose, CA, USA, 2012.
6. Dassault. Cecilia OCAS framework, 2014.
7. F.-X. Dormoy. Scade 6 a model based solution for safety critical software development. In *Embedded Real-Time Systems Conference (2008)*, 2008.
8. M. Güdemann and F. Ortmeier. Model-based multi-objective safety optimization. In *SAFECOMP*, pages 423–436, 2011.
9. N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, Norwell, MA, USA, 1992.
10. S. Humbert, C. Seguin, C. Castel, and J.-M. Bosc. Deriving safety software requirements from an altarica system model. In *SAFECOMP*, pages 320–331, 2008.
11. T. Karnik, P. Hazucha, and J. Patel. Characterization of soft errors caused by single event upsets in cmos processes. *IEEE Trans. Dependable Secur. Comput.*, 1(2):128–143, Apr. 2004.
12. C. Kehren, C. Seguin, P. Bieber, C. Castel, C. Bougnol, J. p. Heckmann, and S. Metge. Architecture patterns for safe design. In *AAAF 1st Complex and Safe Systems Engineering Conference*, 2004.
13. D. Le Berre and A. Parrain. The sat4j library, release 2.2 system description. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:59–64, 2010.
14. E. J. Lefferts, F. L. Markley, and M. D. Shuster. Kalman filtering for spacecraft attitude estimation. *Journal of Guidance, Control, and Dynamics*, 5(5):417–429, 1982.
15. S. Mitra, N. Seifert, M. Zhang, Q. Shi, and K. S. Kim. Robust system design with built-in soft-error resilience. *Computer*, 38(2):43–52, 2005.
16. C. Pagetti, J. Forget, F. Boniol, M. Cordovilla, and D. Lesens. Multi-task implementation of multi-periodic synchronous programs. *Discrete Event Dynamic Systems*, 21(3):307–338, 2011.
17. C. Pagetti, D. Saussié, R. Gratia, E. Noulard, and P. Siron. The ROSACE Case Study: From Simulink Specification to Multi/Many-Core Execution. In *20th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'14)*, April 2014.
18. Y. Papadopoulos and C. Grante. Evolving car designs using model-based automated safety analysis and optimisation techniques. *Journal of Systems and Software*, 76(1):77–89, 2005.
19. C. Preschern, N. Kajtazovic, C. Kreiner, et al. Catalog of safety tactics in the light of the iec 61508 safety lifecycle. In *Proceedings of VikingPLoP 2013 Conference*, page 79, 2013.
20. A.-E. Rugina, P. H. Feiler, K. Kanoun, and M. Kaâniche. Software dependability modeling using an industry-standard architecture description language. In *Embedded Systems and Real-Time Systems (ERTS'08)*, 2008.
21. SAE. Aerospace Recommended Practices 4754a - Development of Civil Aircraft and Systems, 2010.
22. R. A. Shafik, G. Rauwerda, J. Potman, K. Sunesen, D. K. Pradhan, J. Mathew, and I. Sourdis. Software modification aided transient error tolerance for embedded systems. In *Proceedings of the 2013 Euromicro Conference on Digital System Design*, DSD '13, pages 219–226, Washington, DC, USA, 2013. IEEE Computer Society.
23. The Mathworks. *Simulink: User's Guide*. The Mathworks, 2009.
24. M. Walker, M.-O. Reiser, S. T. Piergiovanni, Y. Papadopoulos, H. Lönn, C. Mraidha, D. Parker, D.-J. Chen, and D. Servat. Automatic optimisation of system architectures using east-adl. *Journal of Systems and Software*, 86(10):2467–2487, 2013.
25. P. M. Wells, K. Chakraborty, and G. S. Sohi. Adapting to intermittent faults in multicore systems. *SIGOPS Operating Systems Review*, 42(2):255–264, Mar. 2008.
26. R. Wilhelm and J. Reineke. Embedded systems: Many cores - many problems. In *7th IEEE International Symposium on Industrial Embedded Systems (SIES'12)*, pages 176–180, 2012.