SMT-Based Architecture Modelling for Safety Assessment

Kevin Delmas
ONERA
Toulouse, France
Email: kevin.delmas@onera.fr

Rémi Delmas
ONERA
Toulouse, France
Email: remi.delmas@onera.fr

Claire Pagetti
ONERA
Toulouse, France
Email: claire.pagetti@onera.fr

***Abstract*—Safety is one of the main guidelines for critical systems design. Designers are in charge of developing architectures that comply with the safety requirements. Thus they must ensure that qualitative safety indicators such as the minimal size of failures scenario leading to a failure condition *fc* and quantitative indicators such as the probability to reach *fc* after a certain time interval, are kept below a given threshold. In this paper, we propose efficient minimal cutsets computation and analysis methods fully based on state-of-the-art Satisfiability Modulo Theory (SMT) and Boolean satisfiability (SAT) solvers. The efficiency of minimal cutsets computation, which does not rely on any intermediate model of the system such as *fault trees* is compared to classic formal analysis methods.**

I. Introduction

The design and development of safety critical applications must satisfy stringent dependability requirements. Certification authorities even request the correctness of embedded applications to be proved with formal arguments. To help designers reach that objective, several standards are available. For instance, the aerospace recommended practices (ARP) [1] 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 analysed to check whether the high level safety requirements are fulfilled assuming some properties (such as failure independence, failure modes and propagation rules). In this paper we consider safety requirements based on the following safety indicators: 1) *minimal cardinality of minimal cutsets*: the minimal number of failures before a function loss; 2) *reliability*: the probability of proper function operation over some time period.

Safety assessment is often performed using model-based approaches [2], in which designers build a model of the system's dysfunctional behaviour using an appropriate modelling language. Then this system's model is used to compute the safety indicators of relevant failure conditions using formal methods. Consequently the efficiency and accuracy of these methods is a crucial point of model-based safety assessment.

A. Reminder on dysfunctional modelling and safety assessment

A dysfunctional model (simply called *model*) is a set of interconnected components ensuring several functions. A component is either atomic or composed of sub-components. The classic safety-related terminology used to describe a dysfunctional model is recalled by Definition 1.1.

Definition 1.1: Let C be a component, then: 1) a *failure* F of C is the inability of C to provide a function; 2) a *failure mode* FM of C is the observable effects of the failure F; 3) a *failure condition* FC is $FC = \bigvee_i FM_i$ where FM_i are failure modes. 4) a *failure event* e is the cause of a failure F.

In this paper, we assume that a model satisfies the following properties: 1) **Static**: Reaching a dysfunctional state does not depend on the order of occurrence of basic failure events; 2) **Non repairable**: A failed component cannot return to a working state; 3) **Closed**: The system behaviour depends only on basic failure events; 4) **Exponential law**: The failure events are independent and their probability of occurrence is modelled by an exponential distribution.

Let us remind the basic notions used in safety assessment given in [3]. For static systems, safety assessment is based on the system's *structure function*. This Boolean function over system's failure events indicates if the considered failure condition is true or not. Its circuit representation (*ie* a tree where nodes are logical gates) is the fault tree of the system. The reliability is defined as the probability that the structure function evaluates to false for some operation time value.

Definition 1.2 (Reliability): Let M be a dysfunctional model, e a failure event and t_e the random variable modelling the instant where e occurs. Then the reliability is the probability that e does not occur during time interval $[0, t]$ knowing M is functional at $t = 0$.

$$\begin{aligned}\forall t \in \mathbb{R}^+, R_e(t) \ &\triangleq\ p(t_e > t) = p(\overline{e}) = 1 - p(t_e \leq t)\\ &= 1 - \overline{R_e}(t) = 1 - p(e)\end{aligned}$$

Another safety indicator is the minimal sets of events needed to trigger a given failure condition. For static systems, these sets are the prime implicants of the structure function [4] *i.e* conjunctions of (possibly negated) failure events. The *minimal cutsets* (MCS) are then the restriction of prime implicants to positive literals.

Definition 1.3 (Prime Implicants): Let S be the system model, fc the failure condition and $PI = \{e_1, \cdots, e_k, \neg e_{k+1}, \cdots, \neg e_n\}$ a subset of literals built over system failure events. PI is a prime implicant iff: 1) the conjunction of literals of PI implies fc, 2) $\forall e_j \in PI$, the conjunction $PI \setminus \{e_j\}$ does not imply fc.

In the sequel we refer to the *coherence* (or *monotony*) of a model, that is, for each scenario where the system fails, adding more failure events maintains the failure condition.

978-1-5386-3166-9/17/$31.00 @ 2017 IEEE

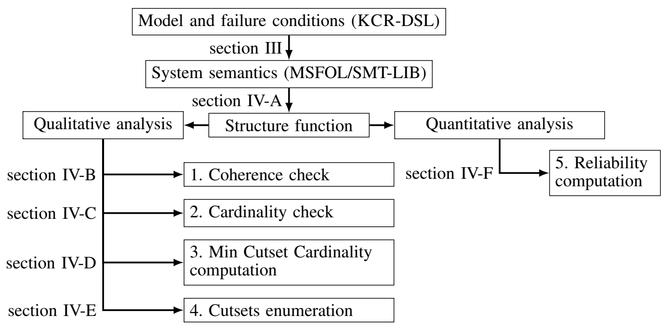


Figure 1: Overall safety assessment flow

This property ensures that the prime implicants of the system's structure function do not contain negative literals, thus that all prime implicants are minimal cutsets.

B. Contribution

The main objective of this paper is to introduce efficient state-of-the-art Satisfiability Modulo Theory (SMT) and Boolean satisfiability (SAT) techniques in model-based safety assessment, following the workflow shown in figure 1. First, the designer describes the system's dysfunctional behaviour in a *domain specific language* (DSL), of our own design, named KCR. The model is then translated into SMT-LIB language [5] allowing to express formula of the *many sorted first order logic* (MSFOL). The structure function of the translated model can then be directly computed by any SMT-lib compatible SMT solver, Microsoft Z3 [6] in our case, and then SAT or SMT solvers can be used to analyse the structure function. As depicted in Figure 1, the followings safety assessments can be performed on a system model *S* and a failure condition *fc*: 1) synthesis of the structure function of *S* wrt. *fc*; 2) checking the coherence of *S* wrt. *fc*; 3) checking the minimal number of failures needed to trigger *fc* against a given threshold; 4) computing the minimal number of failures needed to trigger *fc*; 5) enumerating minimal cutsets for *fc* up to some order k; 6) computing the reliability of the system wrt. *fc* for a given operation time.

To fulfil this objective, we first motivate in Section II the introduction of the safety modelling language KCR. Then Section III gives the formal semantics of KCR by a translation scheme to the SMT-LIB language [5].

The SMT-based assessment methods presented in the figure 1 are then exposed in Section IV. Eventually, Section V provides details about the KCR *analyser* software which implements KCR translation scheme to the SMT-LIB language and all analyses described in the previous sections. Moreover, the KCR analyser's performance for cutsets enumeration is compared to existing assessment tools, on a collection of benchmark systems.

II. Motivation of KCR definition

Let us first explain why yet another modelling language.

A. Dynamic vs static dysfunctional models

As presented in [3], there are two families of dysfunctional models: the static and dynamic ones.

Dynamic systems models A system's model is said *dynamic* when the order of failure/repair occurrences impacts its final failure state. The formalisms used to model such systems (such as Markov chains, Petri nets or Time failure propagation graphs [7]) are usually based on the concepts of state/transition systems and dataflow propagation. The expressiveness of these models allows to describe very complex dysfunctional behaviours of real-world systems. Nevertheless the safety assessment approaches addressing dynamic models (such as stochastic or state space exploration methods) are costly as reminded in [2].

Static systems models Conversely, a system's model is *static* when the order of failure occurrences does not impact its final failure state. Such systems are classically modelled by a collection of Boolean formulae. These formulae are usually inherited from a *failure modes and effects analysis* (FMEA) which describes the conditions under which failure modes are generated and propagated between components in the system. Static modelling is thus less expressive than dynamic one, but is still able to model, or at least approximate, real-world systems. Furthermore, some systems, despite being modelled as dynamic, are static. These systems can be soundly compiled into fault trees [8] and safety assessment can be performed using classic fault tree analysis. The static formalism allows to leverage powerful analytic methods such as Binary Decision Diagrams (BDD) [9], model checking, SMT and SAT to compute minimal cutsets, prime implicants and reliability values.

B. Related works

Given the existing methods recalled previously, an ideal safety-oriented modelling language would: 1) allow to describe systems which are static by construction; 2) facilitate complexity management and model reuse by offering first class components and by separating the specification of safety requirements from that of system components; 3) ease the translation to SMT-LIB problems. Let us now review the most prominent existing languages against these requirements.

Languages for dynamic modelling ALTARICA [10] and its associated tool CECILIA-OCAS [11] is a well-known modelling language used in industrial context. ALTARICA offers the notion of component and failure conditions are defined as special *observer* components composed with the actual system model. In CECILIA-OCAS the minimal sequences enumeration is performed by simulation *i.e* with a complexity of $\mathcal{O}(2^n)$ where n is the number of system's events.

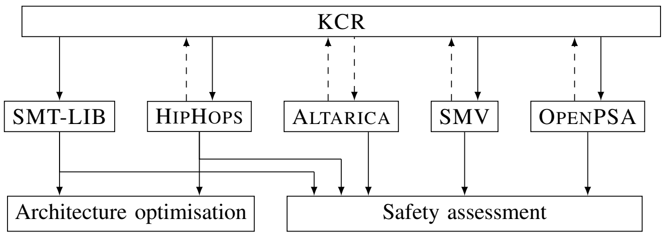
The language SMV [12] and its associated tool XSAP [7] also provides the notion of component and failure conditions are defined as properties. The XSAP tool computes the minimal sequences of the system by a backward or forward symbolic state space exploration presented in [13].

Languages for static modelling The language HIPHOPS [14] models the system as a collection of Boolean formulae. Thus the HIPHOPS underlying formalism is inherently static. These formulae are then processed to generate the structure function of the system. From this point HIPHOPS performs the classic fault tree analysis methods presented in the book [3] of Alain Villemeur. Nevertheless it is important to note that HIPHOPS does not handle first class component which prevents from any model reuse.

Other languages like OpenPSA [15] used by XFTA [16], and the graphical modelling language used by GRIF [17] directly represent the structure function of the system, hence completely loose the architecture of the initial system. Ever since the 90's, most implementations (for instance in GRIF or SAFETY ARCHITECT [18]) rely on BDD to exactly compute system reliability and prime implicants in polynomial time in the BDD size [4]. Nevertheless BDD construction itself could be, in the worst case, exponential in the number of system's basic failure events. Other approaches like the one implemented in XFTA are based on a branch-and-deduce algorithm [16] which performs the enumeration of cutsets directly on the system's fault tree. Intuitively, the brand-and-deduce method first computes an implicant and then minimises it to obtain the minimal cutsets. Other safety assessment methods like [19] address the prime implicants/cutset enumeration of fault trees using SMT solvers.

C. *Motivations of* KCR *definition*

Therefore, existing languages does not fully match our needs of being both a modular and static formalism with an open code base. Instead of developing front-ends to translate existing languages to SMT-LIB, we choose to build a new domain specific language called KCR that would be a reasonable intermediate between ALTARICA for modularity and HIPHOPS for staticity.



The semantics of KCR (based on the MSFOL) is compatible with most of the presented languages. As shown on the above figure, automatic translators (plain arrows) are currently available to HIPHOPS, SMV and OPENPSA. In the future, we aim to offer a full compatibility (dashed arrows) with HIPHOPS, OPENPSA, SMV and ALTARICA (with a staticity check for ALTARICA and SMV models since dynamic systems can not be handled by KCR). These translators should make KCR a pivot language among the presented modelling languages and offer the opportunity to integrate SMT and SAT solvers in the safety assessment process. Moreover this language targets safety-driven architecture optimisation which highly relies on the notion of theory, only available in SMT formalism.

III. MODELLING LANGUAGE

A. KCR *Syntax*

Let us now define the syntax and informal semantics of KCR. We use the traditional Extended Backus-Naur Form (EBNF) language to define the grammatical rules of the language. Let us consider the followings sets: – *ValId* of values identifiers, – *TypeId* of type identifiers, – *CompId* of component identifiers, – *EvtId* of event identifiers, – *InstId* of component instance identifiers, – *FlowId* of flow variable identifiers, – *ConfId* of configuration identifiers.

Let us introduce a macro-expression for lists of elements delimited by a given separator *sep*:

$$\begin{array}{lll l} \mathit{NEList}(x, \mathit{sep}) & ::= & x\ (\mathit{sep}\ x) & \text{non empty list} \\ \mathit{List}(x, \mathit{sep}) & ::= & [\mathit{NEList}(x, \mathit{sep})] & \text{possibly empty list} \end{array}$$

The separator for the syntax is ",", therefore in this section $\mathit{NEList}(x)$ (resp. $\mathit{List}(x)$) is written as a shorthand for $\mathit{NEList}(x, ",")$ (resp. $\mathit{List}(x, ",")$). A model is composed of a system definition and a configuration definition:

$$\mathit{Problem} ::= \mathit{Sys}\ \mathit{Conf}$$

System definition (Sys) is a collection of *type declarations* and *component declarations*. A type declaration (TDecl) defines a set of failure modes values. A component declaration (CDecl) states the functional relationship between input failure modes and internal failure events and the outputs of the component. The keyword `primary` is added if the component is atomic *i.e* does not contain sub-components. Let $t \in \mathit{TypeId},\ v \in \mathit{ValId},\ \mathit{name} \in \mathit{CompId}, e \in \mathit{EvtId}$ then:

$$\begin{array}{lll} \mathit{Sys} & ::= & (\mathit{TDecl})^+\ (\mathit{CDecl})^+ \\ \mathit{TDecl} & ::= & \texttt{type}\ t := \{\mathit{NEList}(v)\}\ ; \\ \mathit{CDecl} & ::= & [\texttt{primary}]\ \texttt{comp}\ \mathit{name}\ (\mathit{List}(\mathit{FDecl})) \\ & & \texttt{returns}\ (\mathit{NEList}(\mathit{FDecl}))\ \{ \\ & & \quad [\ \texttt{evts} : \mathit{NEList}(e)\ ;\ ] \\ & & \quad [\ \texttt{locs} : \mathit{NEList}(\mathit{FDecl})\ ;\ ] \\ & & \quad \texttt{defs} : (\mathit{FDef}\ ;)^+\}\ ; \end{array}$$

Input, output and local variables of a component are called *flow variables* or simply *flows*. A flow f is characterized by (1) *a flow declaration* (FDecl) assigning a type to f; (2) *a flow definition* (FDef) assigning a flow expression F or a component instantiation (with mandatory instance name) to f. Let $f \in \mathit{FlowId},\ t \in \mathit{TypeId},\ c \in \mathit{CompId},\ id \in \mathit{InstId}$ then:

$$\begin{array}{lll} \mathit{FDecl} & ::= & f : t \\ \mathit{FDef} & ::= & f := F \mid \mathit{NEList}(f) := c\ @\ id\ (\mathit{List}(f)) \end{array}$$

A flow variable is to be understood as carrying a *set of failure mode values*, *i.e* an element of the powerset of values of its type (FDecl). Hence, it can carry several values simultaneously or no value at all. The special value *empty* $\in \mathit{ValId}$ represents the absence of failure modes on a flow *i.e* a not faulty flow.

Since flow values are sets of failure modes, flow behaviour can be specified using: 1) the classical set operations (union, intersection, etc.); 2) conditional selection (if-then-else) on Boolean conditions where Boolean expressions are either component's failure events, set membership, or equality/inclusion tests; 3) component instantiation, where output flows of a component instance (identified by an instance identifier $\langle$ `name` $\rangle$), are assigned to a tuple of flow variables. Please note that providing the union operator on flows, which is monotonic wrt. inclusion, allows to easily model coherent systems by defining individual failure modes rules separately and then merge them with the union operator in an output flow. Yet, providing the if-then-else operator still allows to model exclusive failure modes easily. For the sake of readability, a value $v \in \mathit{ValId}$ represents a failure mode in type declarations (TDecl) and the singleton set $\{v\}$ when used in a flow expression. Let $\texttt{inter}, \texttt{union}, ?, =$ be respectively the intersection, union, inclusion, equality over sets and $\#, \&, !$ be respectively the logical OR, AND and NOT connectives. Let $e \in \mathit{EvtId},\ v \in \mathit{ValId},\ f \in \mathit{FlowId}$, then flow expressions are defined as follows:

$$\begin{array}{lll} F & ::= & v \mid f \mid (\ F\ ) \mid F\ \texttt{union}\ F \mid F\ \texttt{inter}\ F \\ & & \mid \texttt{if}\ B\ \texttt{then}\ F\ \texttt{else}\ F \\ B & ::= & e \mid F = F \mid B\ \&\ B \mid B\ \#\ B \mid !B \mid (B) \mid F\ ?\ F \end{array}$$

Example 3.1: Let us consider a system C composed of a sensor S sending data to a control law L. The sensor and law can omit data (failure mode LOST). The KCR model is:

```
type t := {LOST};
comp Sensor() returns (out: t) {
        evts: l; defs: out := if l then LOST else empty;};
comp Law(in:t) returns (out: t) {
        evts: l; defs: out := if l then LOST else in;};
comp C() returns (out: t) { locs: f: t; defs:
f := Sensor@S(); out := Law@L(f);};
```

Configuration definition The safety requirements and analysis parameters are specified by a *configuration* construct identified by a name `cid`. A configuration specifies: `root` – the system's root component; `failure condition` – the *failure condition* as a Boolean expression over system outputs; `duration` – the operation time of the system; `mincard` – the minimal cutsets cardinality requirement; `reliability` – the reliability requirement; `lambdas` – for each atomic component (specified by a $Path$ identifier), the constant failure rate of internal events. Let $rootId \in CompId$, $e \in EvtId$, $name \in InstId, cid \in ConfId$ then:

$$\begin{array}{lll} Conf & ::= & \texttt{config}\ cid\ \{\ \texttt{root} := rootId; \\ & & \texttt{failure condition} := B; \\ & & \texttt{duration} := Int\ ; \\ & & [\texttt{mincard} := Int\ ;] \\ & & [\texttt{reliability} := Real\ ;] \\ & & \texttt{lambdas}\ \{ \\ & & \quad List(Path \rightarrow (NEList(e = Real)\})\};\}; \\ Path & ::= & name\ (.\ name)^* \end{array}$$

The root component must have no inputs (*closed system* hypothesis) and the failure condition must only involve outputs of the root component. Path identifiers $Path$ allow to locate component instances in the root component's *call tree*. That is, a tree in which vertices represent component instances and edges represent component instantiations. A call tree is formally viewed as a set of edges $CT = \{(V_1, V_2) \in InstId^2 \mid V_2 \text{ calls } V_1\}$. Syntactically, a $Path$ identifier is a dot ('.') separated list of instance identifiers, representing the path to the targeted component instance in the root component's call tree. Using this configuration construct, one can thus specify several requirements and analysis parameters for a same system without modifying the system's model in place.

Example 3.2: Going back to the example defined in example 3.1, the root component is `C`, the failure condition is *loss of law output in root component C*, the system operation time is 10 hours, the failure rate of events `l` in component instances `S` and `L` is $10^{-2}.h^{-1}$. Let us assume that single points of failure are prohibited and that the reliability must be greater or equal to 0.8 after 10 hours. The corresponding configuration is:

```
config Conf { root := C; failure condition:= out = LOST;
        duration := 10; mincard:=2; reliability:=0.8;
        lambdas { S → Sensor(l = 0.01), L → Law(l = 0.01) };};
```

B. Overview of SMT-LIB

As said in Section I-B, the KCR semantics is defined by a function Tr which translates KCR to the fragment of Uninterpreted Functions and fixed size BitVectors theories (UFBV) combined with the theory of algebraic datatypes. This logic was chosen because 1) Bitvectors of size n allow to model finite sets and subsets of cardinality at most n, as well as associated operations such as union, intersection and inclusion/membership tests, which captures the intended KCR semantics; 2) KCR is also used to model architecture optimization problem based on safety specific theories.

The UFBV formulae generated by the translation of KCR and the SMT problems used for safety assessment are expressed using the SMT-LIB language [5]. SMT-LIB is a standardized language, understood by a large number of solvers, for expressing SMT problems. These SMT problems are **S**atifiability problems over a set of logic formula **M**odulo a set of first order **T**heories, usually in some decidable fragment of *many sorted first order logic* (MSFOL)[20]. An SMT problem contains: 1) sort and function definitions (said interpreted), 2) function declarations (said uninterpreted) representing the unknowns of the satisfiability problem, 3) assertions *i.e* the formula which must be satisfied.

A sort can be seen as a type, some built-in sorts are provided by SMT-LIB, in our case Boolean and bitvectors of arbitrary yet fixed size n. User-defined sorts can be constructed from built-in sorts with the `define − sort` command. For example an 8-bit bitvector sort `Bv8` is defined as:

```
(define−sort BV8 (_ BitVec 8))
```

Each function must be total, that is, defined for each elements of its domain. In the sequel, a function without inputs is called a *constant* and a function producing a Boolean value is called a *predicate*. Notice that MSFOL functions always return a single output, which can however belong to a *tuple sort*. Tuple sorts are declared as `datatypes` as shown in the following declaration which adds to the problem: 1) `Tuple2`: a sort representing pairs of `BV8`; 2) `mkTuple2`: a constructor for `Tuple2` values taking a pair of `BV8`; 3) `fi`: getter functions returning the i-th field of a `Tuple2`.

```
(declare−datatypes () ((Tuple2 (mkTuple2 (f1 BV8) (f2 BV8) ))))
```

Interpreted functions are defined using (`define − fun`). For instance, a function `BV8_inter` which bitwise-and two `BV8` values is defined by:

```
(define−fun BV8_inter ((a BV8) (b BV8)) BV8 (bvand a b))
```

Uninterpreted functions are declared using (`declare − fun`). For instance, a function `f` taking a single `BV8` and returning a `Tuple2` is declared as:

```
(declare−fun f (BV8) Tuple2)
```

An assertion a is a Boolean formula added to the problem by a statement (`assert a`). A formula is any Boolean expression composed of predicate application, Boolean connectives, *term* equalities and quantifiers:

```
(forall ((x1 T1)...(xn Tn)) b) (exists ((x1 T1)...(xn Tn)) b)
```

A `forall` (resp. `exist`) expression is true whenever `b` is true for all (resp. for some) $x1, \ldots, xn$; A term is a constant `v` or a function application (`f t1...tn`) where each argument is a term of appropriate sort with respect to the function signature.

In the following sections, the modelling of failure modes generation and propagation is based on the *let binder* construct which defines an expression `e` built from identifiers `x1, . . . , xn` representing auxiliary expressions `e1, . . . , en`; and on the built-in *conditional selection* which stands for *if* `b` *then* `t` *else* `e`.

```
(let ((x1 e1)...(xn en)) e) (ite b t e)
```

Solving an SMT problem means *finding a total assignment of the unknowns of the problem which satisfies the assertions*, this assignment is called a *model* of the problem. The (`check − sat`) command asks the solver to find a model of the problem. If the answer is SAT, the model is obtained by the (`get − model`) command. Otherwise the problem is UNSAT and an unsatisfiability proof can, optionally, be generated.

C. System semantics

Before exposing the semantics of KCR, let us note that SMT-LIB separator for list elements is the "␣" character, so in the sequel we will write $NEList(x)$ (resp. $List(x)$) for $NEList(x, "␣")$ (resp. $List(x, "␣")$).

The semantics of a KCR system is given by the semantics of the translation of its type and component definitions. Let $Sys = TDecl_1 \cdots TDecl_n \ CDecl_1 \cdots CDecl_m$ so:

$$\begin{array}{lll} Tr(Sys) & \triangleq & Tr(TDecl_1) \cdots Tr(TDecl_n) \\ & & Tr(CDecl_1) \cdots Tr(CDecl_m) \end{array}$$

Types semantics $(TDecl)$ A type t is translated as a bitvector sort of size $card(t)$. Each constant v_i of type t is translated as a constant bitvector where all bits except the i-th are set to 0. The particular *empty* value is the bitvector where all bits are zero. Let $TDecl = \texttt{type}\ t := \{v_1, \cdots, v_n\}$; where $v_i \in ValId,\ t \in TypeId$ and x_i be the string of size n representing the binary value 2^{i-1} if $i \in [1, n]$ else 0, then:

$$\begin{array}{lll} Tr(TDecl) & \triangleq & (\texttt{declare-sort}\ t\ (\_\ \texttt{BitVec}\ n)) \\ & & (\texttt{define-const}\ empty\ Tr(t)\ \#\text{b}x_0) \\ & & (\texttt{define-const}\ v_1\ Tr(t)\ \#\text{b}x_1) \\ & & \cdots (\texttt{define-const}\ v_n\ Tr(t)\ \#\text{b}x_n) \end{array}$$

Component semantics $(CDecl)$ A *component declaration* is translated as an interpreted function which takes 1) a tuple of bitvector sorts corresponding to its input flow variables, 2) a tuple of Booleans corresponding to its failure events, and returns a tuple of bitvector sorts corresponding to its output flows. The signature of the component functions is extended recursively with their sub-components failure events. To avoid name conflicts, the events are renamed using the renaming function rnm which prefixes a failure event with the instance alias of its enclosing component *i.e* $rnm(\texttt{compId@id}, \texttt{e}) = \texttt{id.e}$. The local flow declarations $FDecl_{l_i} = l_i : t_i$ are only used to type-check the flow definitions $FDef_{l_i} = l_i := F_i$. Then local declarations are dropped and only definitions are kept to produce the flow definition of output flow o_i. Let $List((Tr(e')\ \texttt{Bool}))$ be the failure events list inlined from sub-components, $\texttt{Tuple}M$ be the sort of M-sized tuples and $t_1, \cdots, t_m$ be the sorts of $o_1, \cdots, o_m$ then:

$$\begin{array}{lll} Tr(CDecl) & \triangleq & (\texttt{define-fun}\ name \\ & & \left.\begin{array}{l} (List(Tr(FDecl)) \\ List((Tr(e)\ \texttt{Bool})) \\ List((Tr(e')\ \texttt{Bool}))) \end{array}\right\}\text{inputs} \\ & & (\texttt{Tuple}M\ Tr(t_1) \cdots Tr(t_m))\ \}\text{output} \\ & & (Tr((FDef;)^+))\ \}\text{body} \end{array}$$

Flow definition semantics $(FDef)$ Let $FDef_1; \cdots FDef_n$; where $FDef_i = f_i{:=}F_i$; with $f_i \in FlowId$ and F_i is a flow expression. Let us consider that the m last flow definitions define output flows then

$$\begin{array}{lll} Tr(FDef_1; \cdots FDef_n;) & \triangleq & (\texttt{let}\ Tr(FDef_1) \\ & & \cdots (\texttt{let}\ Tr(FDef_n) \\ & & (\texttt{mkTuple}\ Tr(f_{n-m}) \\ & & \cdots Tr(f_n)) \cdots) \end{array}$$

A flow definition is given either by an expression F *i.e* $f := F$, in that case $Tr(FDef) \triangleq (Tr(f)\ Tr(F))$; or by a component instantiation where a flow identifier must be given for each component output. Let $FDef = List(f) := c\ @\ name\ (List(f'))$ where $f, f' \in FlowId,\ c \in CompId$ and $I = List(Tr(f'))\ List(Tr(e'))$ where $List(e')$ are the inlined events of *name* and i is the index of f in $List(f)$ then

$$Tr(FDef) \quad \triangleq \quad List((Tr(f)\ (\texttt{field}_\texttt{i}\ (Tr(c)\ I))))$$

Flow declaration semantics $(FDecl)$ The semantics of a *flow declaration* is an association of a type and a flow identifier thereafter used in the `let` structure. Let $FDecl = f : t$ where $f \in FlowId,\ t \in TypeId$ then:

$$Tr(FDecl) \triangleq (Tr(f)\ Tr(t))$$

Flow expression semantics (F) A flow is defined by an expression F which can contain Boolean expressions B. These expressions are translated recursively to MSFOL expressions using the Tr function. We remind that a failure mode (not to confuse with failure events which are uninterpreted Boolean constants) is encoded as a bitvector with only one bit set to 1. A set of failure modes is encoded as a bitvector in which the i^{th} is true iff the failure mode i of the corresponding type belongs to the set. The union (resp. intersection) of two sets is achieved by a bitwise-or (`bvor`) (resp. bitwise-and `bvand`). Let $e \in EvtId,\ v \in ValId,\ f \in FlowId,\ t \in TypeId,\ c \in CompId,\ f \in FlowId$ then:

$$\begin{array}{lll} Tr(v) & \triangleq & \texttt{v of sort } Tr(t) \\ Tr(c) & \triangleq & \texttt{c function identifier} \\ Tr(e) & \triangleq & \texttt{e of sort Bool} \\ Tr(t) & \triangleq & \texttt{t bitvector sort identifier} \\ Tr(f) & \triangleq & \texttt{f of sort } Tr(t) \\ Tr(!\ B) & \triangleq & (\texttt{not}\ Tr(B)) \\ Tr(B_1\ \&\ B_2) & \triangleq & (\texttt{and}\ Tr(B_1)\ Tr(B_2)) \\ Tr(B_1\ \#\ B_2) & \triangleq & (\texttt{or}\ Tr(B_1)\ Tr(B_2)) \\ Tr(F_1\ \texttt{union}\ F_2) & \triangleq & (\texttt{bvor}\ Tr(F_1)\ Tr(F_2)) \\ Tr(F_1 = F_2) & \triangleq & (=\ Tr(F_1)\ Tr(F_2)) \\ Tr(F_1\ \texttt{inter}\ F_2) & \triangleq & (\texttt{bvand}\ Tr(F_1)\ Tr(F_2)) \\ Tr(F_1\ ?\ F_2) & \triangleq & Tr((F_1 \cap F_2) = F_1) \\ Tr(\texttt{if}\ B\ \texttt{then}\ F_1\ \texttt{else}\ F_2) & \triangleq & (\texttt{ite}\ Tr(B)\ Tr(F_1)\ Tr(F_2)) \end{array}$$

Example 3.3: Consider the system described in example 3.1. Its translation to SMT-LIB is the following:

```
(declare−datatypes (T) ((Tuple1 (mkTuple1 (field1 T)))))
(declare−sort t (_ BitVec 1))
(define−const LOST t #b1) (define−const empty t #b0)
(define−fun C ((S.l Bool) (L.l Bool)) (Tuple1 t)
                    (let ((f (Sensor S.l)) (mkTuple1 (Law f L.l))))
(define−fun Sensor ((l Bool)) (Tuple1 t)
                    (mkTuple1 (ite l LOST empty)))
(define−fun Law ((in t) (l Bool)) (Tuple1 t)
                    (mkTuple1 (ite l LOST in)))
```

Configuration semantics The *failure condition* given by $failure\ :=\ B$ is translated as a predicate over system's outputs. Note that *failure* is the only configuration field translated into MSFOL, since other fields are numerical parameters for analyses. Let $o_1, \cdots, o_n$ be the outputs of `sys` then:

$$\begin{array}{lll} Tr(failure\ :=\ B) & \triangleq & (\texttt{define-fun fail} \\ & & ((Tr(o_1)\ Tr(t_1)) \\ & & \cdots (Tr(o_n)\ Tr(t_n))) \\ & & \texttt{Bool}\ Tr(B)) \end{array}$$

Example 3.4: The failure condition of example 3.1 is encoded as a predicate over system `C`'s outputs.

```
(define−fun fail ((out t)) Bool (= out LOST))
```

IV. Safety assessment

The MSFOL model encoding we just defined allows to use off-the-shelf SMT solvers to achieve safety assessments expressed as satisfiability checks. In the following section, we detail the SMT problems allowing to 1) check the coherence of a system (section IV-B); 2) compute the system reliability (section IV-F); 3) compute the minimal cardinality of minimal cutsets (section IV-D).

A. Computing the structure function

For static systems the *structure function* φ is the composition of the failure condition and the system flow function. In our case both are obtained by a translation to the UFBV logic of a KCR model. Therefore, we use the UF theory for synthesising the structure function φ. To do so we ask Z3 the following question: *Is there a model of φ (`phi`) satisfying Definition 4.1 ?*

Definition 4.1 (Structure function): Let `rootId` $= Tr($`rootId`$)$ (the system) and `fail` $= Tr(failure := B)$ (the failure condition). Then the structure function `phi` is an uninterpreted SMT-LIB function which satisfies the following assertion:

```
(declare−fun phi (Bool ... Bool) Bool)
(assert (forall ((e1 Bool) ... (en Bool))
    (= (phi e1 ... en)
        (fail (field1 (rootId e1 ... en))...(fieldm (rootId e1 ... en))))))
```

Example 4.1: The structure function example 3.1 computed by Z3 is then:

```
(define−fun phi ((S.l Bool) (L.l Bool)) Bool (or S.l L.l))
```

B. Checking coherence

Verifying the *coherence* of a system boils down to check the monotonicity of `phi`[3], which is achieved by the following satisfiability check:

Query 1 (Coherence): Let `phi` be the structure function of a system for a given failure condition, then the system is coherent iff the following problem is UNSAT:

```
(assert (exists ((e1 Bool) ... (en Bool) (e'1 Bool) ... (e'n Bool))
(and (and (=> e1 e'1) ... (=> en e'n)))
    (not (=> (phi e1 ... en) (phi e'1 ... e'n)))))
```

C. Checking the cardinality of the shortest minimal cutset

As mentioned in [4], the *minimal cutsets* (MCS) can be defined as the restriction of prime implicants of the structure function to positive literals. Thus the cardinality of the shortest MCS is the minimal number of positive literals found in the prime implicants of the structure function obtained by Definition 4.1. Therefore checking the system compliance for cardinality requirement k boils down to prove that it does not exist any cutset containing more than k events.

The encoding of this problem is based on an operator (called *population count*) counting the number of true Booleans among a tuple of Booleans. One can use one of the efficient population count encodings which have been studied to great length, for instance in [21]. Note that the approach given in Definition 4.2 requires neither computing explicitly the prime implicants of φ, nor computing the minimal prime implicants cardinality.

Definition 4.2 (Cardinality check): Let `rootId` be the system function over $e_1, \cdots, e_n$ failure events, `fail` the failure condition function, $\texttt{atMost}_\texttt{k}$ an implementation of population count and k be the minimal cardinality requirement. The system is compliant iff the following problem is UNSAT.

```
(declare−const e1 Bool)...(declare−const en Bool)
(assert (fail (field1 (rootId e1 ... en))...(fieldm (rootId e1 ... en))))
(assert (atMost_k e1 ... en))
```

D. Computing the cardinality of the shortest minimal cutset

We recall that the cardinality of the smallest MCS is a classic qualitative safety indicator since it represents the minimal number of failures before a function loss. The cardinality of the smallest MCS is classically computed from a BDD of the structure function [4]. However, this computation assumes that the BDD of the structure function is computable. Experiments of Section V contain some systems where the BDD computation is not feasible in reasonable time. We propose to use our SMT encoding to compute the cardinality of the shortest cutset of MCS. To do so, we reuse the cardinality check problem and increment k from $k = 0$ until the solver answers SAT. The last value of k is then the size of the shortest MCS. Thus if one is just interested in the lower bound of MCS size, this method provides a simple way to compute it without computing all MCS beforehand.

Example 4.2: For $k = 0$ the solver proves that the failure condition of example 3.1 cannot be triggered; at $k = 1$ the problem is SAT, so the cardinality of the smallest MCS is 1.

E. Enumerating minimal cutsets up to order k

Some safety processes like Failure Modes and Effects Analysis (FMEA) require to enumerate minimal cutsets up to a fixed size k. Therefore, we propose an iterative enumeration method based on either SMT or SAT model enumeration method of the problem introduced in Definition 4.2. Starting with $i = 0$, a cardinality constraint $\Sigma_j e_j \leq i$ over system's failure events e_j is added (in the first iteration) or replaced (in subsequent iteration) in the problem, to ensure that each found model contains at most i true literals. Each found model is restricted to its positive literals, saved apart as a cutset, and its negation is added as a blocking clause. The restricted model is indeed a cutset, since any model containing less true literals would have been found and blocked by a blocking clause in a previous or the same iteration. For each cutsets size i, the problem's models are enumerated until the instance becomes UNSAT. Then i is incremented and the whole loop is iterated until i reaches the user-specified bound k.

Since the structure function is purely boolean, this algorithm can also be implemented with a SAT solver, by translating the cardinality constraint to clauses [22]. The advantage of SAT over SMT solvers are: 1) more efficient model enumeration algorithms (`sharpCDCL` [23]); 2) multicore parallelism (`clingo` [24]).

Example 4.3: Let us compute the MCS of size one or less of the structure function found in the example 4.1. For

$k = 0$ the problem is UNSAT so it does not exist cutsets of size 0 for φ; at $k = 1$ the problem has two models $\{\{S.l, \neg L.l\}, \{\neg S.l, L.l\}\}$ which gives two cutsets $\{S.l\}$ and $\{L.l\}$.

F. Computing reliability

Concerning reliability, we translate the structure function synthesised by the SMT solver to a BDD and use the classic BDD-based method given in [25]. The top event probability (*i.e* the *unreliability*) is computed in a top-down fashion on the BDD. The computation cost is linear in the BDD size. Let F be a formula, v the variable of the root node of the BDD of F , F_1 resp. F_0 the BDD of the formula F where $v = 1$ resp. $v = 0$ then the top event probability is $p(v.F_1 + \overline{v}.F_0) = p(v).p(F_1) + [1 - p(v)].p(F_0)$

Example 4.4: The reliability of the example 3.1 is computed with $\lambda_{L.l} = \lambda_{S.l} = 0.01$ and operating time $t = 10$. $R = 1 - [p(S.l) + [1 - p(S.l)].p(L.l)] = e^{-0.2}$

V. IMPLEMENTATION AND EXPERIMENTS

The following section introduces the KCR analyser tool and provides a comparison of computation times for minimal cutset enumeration between our SAT-based approach and the safety assessment tools XSAP, XFTA, HIPHOPS and GRIF.

A. The KCR *analyser*

The KCR language and associated SAT/SMT-based analyses presented in this paper are fully implemented in a tool named KCR *analyser* available at [26] with associated benchmarks. Written in SCALA, it is built on proven third party libraries: ANTLRv4 [27] for parsing, the Z3 SMT solver [6] through its Java API for all SMT duties (saving us the reimplementation of translating quantified bitvectors formulas and algebraic datatypes to propositional logic), and JavaBDD [28] for BDDs. Thus, the analyser consists only of KCR-to-SMT translation, KCR-to-BDD translation, orchestration of satisfiability checks, and is hence relatively small.

B. The benchmarks

Experiments are performed on a collection of four benchmarks (available in the tool distribution): 1) the ROSACE case study extracted from [29], model of a longitudinal controller of a medium-range aircraft; 2) the FUEL system given as example with the HIPHOPS tool[1]; 3) the HBS system, model of a hybrid breaking system, fully explains on HIPHOPS website[1]; 4) the QUADCOPTER system, model of a semi automatic drone navigation manager, presented in [30].

Furthermore we add a synthetic example called GRID, whose components are simple sources and transmitters which can only be on a working or failed state. More precisely the GRID system is a matrix of 3×60 transmitters (with a first column of sources) where each component transmits its output to its east, south-east and north-east neighbours. A component transmits a correct value if at least one of its inputs is correct and if the component is functional. The grid fails if all values received at the end of the grid are incorrect. GRID is used to compare the MBSA tools to GRIF on a example modelled with reliability block diagram formalism.

[1]http://hip-hops.eu/index.php

	k	HIPHOPS	XSAP	XFTA	SAT
ROSACE	1	0.383	> 100	0.045	**0.035**
	2	0.426	> 100	**0.048**	0.119
	3	0.489	> 100	**0.047**	0.287
	4	0.645	> 100	**0.051**	1.069
	7	9.408	> 100	**0.298**	> 100
FUEL	1	> 100	> 100	**0.047**	0.062
	2	> 100	> 100	**0.063**	0.141
	3	> 100	> 100	**0.302**	0.314
	4	> 100	> 100	2.678	**0.743**
	7	> 100	> 100	> 100	**13.404**
HBS	1	21.552	> 100	0.073	**0.044**
	2	23.236	> 100	**0.064**	0.322
	3	34.494	> 100	**0.154**	0.735
	4	57.828	> 100	**0.948**	2.193
	7	> 100	> 100	84.797	**59.786**
QUADCOPTER	1	17.511	> 100	0.039	**0.026**
	2	22.372	> 100	**0.038**	0.044
	3	37.088	> 100	**0.039**	0.076
	4	> 100	> 100	**0.039**	0.263
	7	> 100	> 100	**0.038**	2.892

Table I: Execution time for cutsets enumeration

C. Experiments on MCS computation time

We compare our SAT-based method with a single thread implementation and trial versions of third party tools HIPHOPS, XFTA and XSAP. These execution times have been obtained on a Intel XeonE5-2609 v2.0@2.50GHz (8 cores), 64GB RAM. In all the tables, the computation times for cutsets enumeration is given for several maximum cutsets sizes k and the most efficient time is in bold.

Case studies - Table I As expected the experiments shows that cutsets enumeration on the SMV models of the benchmarks is way slower than other enumeration approaches. This is due to the state space exploration which does not scale well for large systems.

Concerning HIPHOPS, the experiments show that the cutsets enumeration based on the classic fault tree analysis does not scale for the largest benchmarks. Indeed the cutsets enumerations performed by our SAT approach and XFTA are often, by several orders of magnitude, faster than HIPHOPS.

On most of the benchmarks, the XFTA tool performs the fastest cutsets enumeration among the considered tools. Note that in most cases our SAT-based approach execution time is in the same order of magnitude than the XFTA one. Indeed XFTA uses a SAT-like branch-and-deduce algorithm close to the one used by the SAT solver to find a model of the problem presented in the section IV-E. But for systems with thousands of cutsets (like these benchmarks) adding a conflict clauses per found cutset becomes the main limitation of our SAT-based approach, which is not the case for the brand-and-deduce algorithm of XFTA. Nevertheless the advantage of our SAT-based approach is illustrated on FUEL and HBS systems.

Case studies - Table II The table II compares the computation times of GRIF to other tools. The results confirm the above observation and show the limitations of BDD-based approach used by GRIF for the enumeration of cutsets of large systems.

	k	GRIF	HIPHOPS	XSAP	XFTA	SAT
GRID	3	0.30	> 100	> 100	**0.063**	0.413
	4	1.154	> 100	> 100	**0.097**	1.431
	5	22.714	> 100	> 100	**0.143**	10.206
	6	> 100	> 100	> 100	**0.293**	56.459

Table II: Execution time for GRID cutsets enumeration

VI. Conclusion

Summary The KCR DSL presented in section III allows to describe static systems and to specify the associated safety specifications. Moreover the KCR semantics is explicitly defined by translation to *many sorted first order logic* which gives the opportunity to use efficient SMT or SAT solvers to deal with safety assessment. We explained in section IV how to encode safety assessment as simple SMT or SAT queries. The safety requirements as MCS cardinality computation is directly performed by a third-party SMT solver without using BDD representation nor fault trees. We defined an SMT problem allowing us to assess whether a system satisfies MCS cardinality requirements without explicitly computing the MCS. Last, we proposed an SMT problem allowing to check the coherence property on a system. We show that SMT and SAT outperform BDD based cutsets enumeration methods in some cases, and open the way to architecture synthesis.

Ongoing and future works The methods presented in this paper are meant as building blocks for design space exploration. Design space exploration of fault-tolerant systems, under qualitative constraints has been addressed in several previous papers like [31]. Most of these earlier approaches use genetic algorithms to explore the design space. These algorithms breed numerous alternative evolutions of an initial architectural design while continuously assessing their fitness according to qualitative fault-tolerance properties. Our ongoing work on SAT and SMT-based design space exploration, initiated in [32], can reduce the exploration time obtained with genetic approaches by reducing the cost of evaluating each candidate design, and by definitively rejecting unsatisfying solutions during exploration (through conflict learning).

References

[1] SAE, "Aerospace Recommended Practices 4754a - Development of Civil Aircraft and Systems," 2010.

[2] A. Rauzy and C. Bleriot-Fabre, "Model-based safety assessment: Rational and trends," in *Mecatronics-2014-Tokyo*. IEEE, 2014, pp. 1–10.

[3] A. Villemeur, *Reliability, availability, maintainbility and safety assessment*. John Wiley & Sons, 1992.

[4] A. Rauzy, "Mathematical foundations of minimal cutsets," in *IEEE Transaction on Reliability*, vol. 50, no. 4, december 2001.

[5] C. Barrett, A. Stump, and C. Tinelli, *The SMT-LIB Standard Version 2.0*, University of Iowa, December 2010.

[6] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[7] B. Bittner, M. Bozzano, R. Cavada, A. Cimatti, M. Gario, A. Griggio, C. Mattarei, A. Micheli, and G. Zampedri, "The xsap safety analysis platform," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2016, pp. 533–539.

[8] A. Rauzy, "Mode automata and their compilation into fault trees," *Reliability Engineering & System Safety*, vol. 78, no. 1, pp. 1–12, 2002.

[9] R. E. Bryant, "Symbolic boolean manipulation with ordered binary-decision diagrams," *ACM Computing Surveys (CSUR)*, vol. 24, no. 3, pp. 293–318, 1992.

[10] A. Arnold, G. Point, A. Griffault, and A. Rauzy, "The altarica formalism for describing concurrent systems," *Fundamanta Informaticae*, vol. 40, no. 2-3, pp. 109–124, 1999.

[11] Dassault Aviation, *Cecilia Workshop: User's Manual*, Dassault Aviation, 2011.

[12] M. Bozzano, A. Cimatti, and C. Mattarei, "Automated analysis of reliability architectures," in *2013 18th International Conference on Engineering of Complex Computer Systems, Singapore, July 17-19, 2013*. IEEE Computer Society, 2013, pp. 198–207.

[13] M. Bozzano, A. Cimatti, and F. Tapparo, "Symbolic fault tree analysis for reactive systems," in *International Symposium on Automated Technology for Verification and Analysis*. Springer, 2007, pp. 162–176.

[14] Y. Papadopoulos and J. A. McDermid, "Hierarchically performed hazard origin and propagation studies," in *18th International Conference on Computer Safety, Reliability and Security*, 1999.

[15] E. Steven and R. Antoine, *Open-PSA Model Exchange Format*, February 2017.

[16] A. Rauzy, "Xfta: pour que cent arbres de défaillance fleurissent au printemps," *Actes du Congrès Lambda-Mu*, vol. 18, 2012.

[17] *GRIF 2016 Fault Tree User Manual*, Total, January 2016.

[18] J. Dumont, F. Sadmi, and F. Vallee, "Safety architect: un outil d'analyse de risques s'inscrivant dans les processus d'ingénierie de systèmes complexes," *Génie logiciel*, vol. ISSN 1265-1397, no. 98, pp. 27–33, 2011.

[19] J. Kühn and al, "Safety conflict analysis in medical cyber-physical systems using an smt-solver," in *Gemeinsamer Tagungsband der Workshops der Tagung Software Engineering 2015, Dresden, Germany, 17.-18. März 2015.*, ser. CEUR Workshop Proceedings, vol. 1337. CEUR-WS.org, 2015, pp. 19–23.

[20] C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli, "Satisfiability modulo theories." *Handbook of satisfiability*, vol. 185, pp. 825–885, 2009.

[21] A. M. Frisch and P. A. Giannaros, "Sat encodings of the at-most-k constraint. some old, some new, some fast, some slow," in *Proc. of the Tenth Int. Workshop of Constraint Modelling and Reformulation*, 2010.

[22] T. Philipp and P. Steinke, "Pblib - A library for encoding pseudo-boolean constraints into CNF," in *Theory and Applications of Satisfiability Testing - SAT 2015 - 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings*, 2015, pp. 9–16.

[23] V. Klebanov, N. Manthey, and C. J. Muise, "Sat-based analysis and quantification of information flow in programs," in *Quantitative Evaluation of Systems - 10th International Conference, QEST 2013, Buenos Aires, Argentina, August 27-30, 2013. Proceedings*, 2013, pp. 177–192.

[24] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub, "Clingo = ASP + control: Preliminary report," *CoRR*, vol. abs/1405.3694, 2014. [Online]. Available: arxiv.org/abs/1405.3694

[25] A. Rauzy, "New algorithms for fault trees analysis," *Reliability Engineering and System Safety*, vol. 40, pp. 203–211, 1993.

[26] "KCR analyser download page," www.onera.fr/en/staff/kevin-delmas?page=1.

[27] T. Parr, *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013.

[28] "The JavaBDD library," javabdd.sourceforge.net/.

[29] C. Pagetti, D. Saussié, R. Gratia, E. Noulard, and P. Siron, "The rosace case study: From simulink specification to multi/many-core execution," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*. IEEE, 2014, pp. 309–318.

[30] T. Prosvirnova, J. Brunel, and C. Seguin, "Performing safety analyses with aadl and altarica," under submission, 2017.

[31] M. Adachi, Y. Papadopoulos, S. Sharvia, D. Parker, and T. Tohdo, "An approach to optimization of fault tolerant architectures using hip-hops," *Software: Practice and Experience*, vol. 41, no. 11, pp. 1303–1327, 2011.

[32] K. Delmas, R. Delmas, and C. Pagetti, "Automatic architecture hardening using safety patterns," in *Computer Safety, Reliability, and Security*. Springer, 2015, pp. 283–296.