ALPACAS: A Language for Parametric Assessment of Critical Architecture Safety

Maxime Buyse
Uber Elevate, Paris, France

Rémi Delmas
Uber Elevate, Paris, France

Youssef Hamadi
Uber Elevate, Paris, France

Abstract

This paper introduces ALPACAS, a domain-specific language and algorithms aimed at architecture modeling and safety assessment for critical systems. It allows to study the effects of random and systematic faults on complex critical systems and their reliability. The underlying semantic framework of the language is Stochastic Guarded Transition Systems, for which ALPACAS provides a feature-rich declarative modeling language and algorithms for symbolic analysis and Monte-Carlo simulation, allowing to compute safety indicators such as minimal cutsets and reliability. Built as a domain-specific language deeply embedded in Scala 3, ALPACAS offers generic modeling capabilities and type-safety unparalleled in other existing safety assessment frameworks. This improved expressive power allows to address complex system modeling tasks, such as formalizing the architectural design space of a critical function, and exploring it to identify the most reliable variant. The features and algorithms of ALPACAS are illustrated on a case study of a thrust allocation and power dispatch system for an electric vertical takeoff and landing aircraft.

2012 ACM Subject Classification Software and its engineering → Domain specific languages; Computer systems organization → Embedded and cyber-physical systems

Keywords and phrases Domain-Specific Language, Deep Embedding, Scala 3, Architecture Modelling, Safety Assessment, Static Analysis, Monte-Carlo Methods

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2021.5

Supplementary Material *Software (ECOOP 2021 Artifact Evaluation approved artifact)*: https://doi.org/10.4230/DARTS.7.2.14

1 Introduction

The work presented in this paper is motivated by the emergence of Urban Air Mobility (UAM) which will move people and cargo by air, exploiting the third dimension to escape ground congestion. UAM will be powered by new electric Vertical Take-Off and Landing (eVTOL) aircraft. They will use highly redundant fully electric propulsion systems for reduced noise and safe operation in urban areas. The Aerospace Recommended Practices (ARP-4754A[1]/4761[2]) guide the design and certification process of these aircraft. According to [18], safety assessment is very challenging for eVTOL development with large costs associated to safety modeling, and difficulties to assess and optimize multiple architecture variants.

New eVTOL companies propose very different system architectures (lift-only configurations, lift+cruise configurations with tilt-wing, tilt-rotor, etc.) for a wide variety of applications (air taxi, deliveries, freight, etc.) and safety aspects play a decisive role in the

[1] https://www.sae.org/standards/content/arp4754a/
[2] https://www.sae.org/standards/content/arp4761/

35th European Conference on Object-Oriented Programming (ECOOP 2021).
Editors: Manu Sridharan and Anders Møller; Article No. 5; pp. 5:1–5:29
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

competition of designs. Moreover, exploring the underlying design-space from a safety and certification perspective can help define meaningful mandatory safety targets, which are still being actively discussed by regulators in the US and the EU.

A system is called *critical* when the failure to perform its function is likely to result in loss of life or extreme environment damage. Examples of critical systems are embedded aircraft control systems, railway control systems, nuclear plant control systems, radiotherapy equipment control systems, etc. The acceptable risk levels for critical systems are defined by competent regulatory bodies, collaboratively with stakeholders such as system providers, system users, the state, etc. The severity of identified risks determines fault tolerance and reliability requirements for the system, as well as design and verification process requirements. *System safety assessment* consists in characterizing the risk for a particular system, identifying applicable safety requirements and demonstrating that the planned system architecture meets safety requirements.

All phases of the safety process are backed by modeling and analysis tasks in some adapted formalism. The modeling artifacts are used as evidence in the certification process. Implementation requirements [11] are derived from the safety analysis to feed the implementation phase following DO-178C and DO-254A recommendations. Similar concepts apply in other domains such as automotive and railway [39] [46].

The safety, verification and validation activities of critical embedded systems account for a large part of the total development cost. Identifying the optimal system architecture according to safety metrics and implementation cost criteria before starting its implementation and certification is hence essential, in particular in the UAM domain where designs are created from a blank slate without preexisting reference or safety record. A lack of agility in these early design phases can result in suboptimal system designs and limit programmatic agility in the long run, i.e. the ability to update an existing system with new functions or safety enhancing features that would require substantial modifications of the safety models and analysis.

As will be seen in the related works section, current safety formalisms lack features which could make safety modeling more efficient. These features are commonly found in modern functional and object-oriented programming languages: encapsulation, generic parameters, higher-order parameters, polymorphism, etc. Better support for incremental and generic modeling can allow to go beyond safety assessment and support genuine safety-driven *design-space exploration*, where optimal design decisions are made by comparing automatically several candidate system architectures. For this we propose the ALPACAS safety formalism, built as an embedded domain-specific language in the Scala 3 functional programming language. ALPACAS offers first-class generic and parametric modeling capabilities allowing to formalize higher-order design spaces. The embedding allows to fuse declarative safety modeling and programming in a coherent framework, to compute safety indicators for system variants more easily, effectively unlocking architectural design-space exploration and optimization.

The rest of the paper is structured as follows: Section 2 reviews existing safety modeling formalisms and their limitations, as well as domain-specific language implementation techniques; Section 3 presents the design goals and requirements that shaped ALPACAS, together with a running example; Section 4 introduces the ALPACAS syntax and implementation using the running example; Section 5 describes the formal semantics of ALPACAS; Section 6 discusses safety analysis algorithms provided by ALPACAS; Section 7 describes a design-space exploration study performed with ALPACAS for a thrust reallocation function of an electric vertical takeoff and landing aircraft; Last, Section 8 concludes the paper and outlines perspectives to this work.

2 Related works

We present core safety modeling concepts in Section 2.1, related works on safety modeling and analysis in Section 2.2, as well as relevant literature on domain-specific language implementation techniques in Section 2.3.

2.1 Core safety concepts

We now review fundamental concepts in system safety modeling as originally presented in [47]. A *System* is an assembly of *Components*, operating together to perform a *Function*. *Basic Failure Events* cause changes of the internal *State* of components. At the very least, a component has two states: *working* and *failed*, but it can have more, such as multiple functional or degraded modes. *Failure Modes* are the external manifestations of the internal failure state of a component. For instance, a valve component could be in three states: *working*, *stuck-open*, *stuck-closed*, with corresponding failure modes *nominal pressure*, *over-pressure* or *under-pressure*, respectively. Failure modes propagate and combine through the system, affecting its ability to perform its function. A *Failure Condition* is a failure mode of the function performed by the system, and it is the consequence of one or more basic failure events. The *Structure Function* of the system specifies how basic failure event combinations or sequences produce different failure conditions at the system level.

Failure events occur randomly following certain delay distributions, failure behaviour can be non-monotonic and sensitive to event ordering, propagations can exhibit some level of randomness and time dependency, which makes safety modeling and analysis a complex problem. Many formalisms have been proposed, depending on the class of system to analyze. In all cases, safety models are built in order to compute *safety indicators* of a system and predict its performance. *Qualitative indicators* describe the logical relationship between basic failure events and system failure conditions. *Minimal cutsets or sequences* (MCS) are minimal event combinations or sequences triggering a failure condition. *Quantitative indicators* capture the probabilistic aspects of system failure. For instance, *Unreliability*, the probability that the system fails in the interval $[0, T]$, depends in a non-trivial way on basic event probabilities and on system architecture.

2.2 Safety formalisms

Safety formalisms are distinguished by their semantics, which delimits the class of real-world systems they can faithfully model. Semantics also influences the tractability of safety indicators. The other major aspect for use in real-world applications is the level of support for design-space exploration, i.e. the ease with which models can be parameterized, updated, extended, reused, etc. Each new system design iteration alters the system architecture and its dysfunctional behaviour, which must be reflected in the safety model. Design modifications are also largely guided by the safety analysis of different design options which orient the choice of fault-tolerance patterns, redundancy levels, basic event occurrence rates, etc.

The most widely used safety formalism in industrial domains are *Fault Trees* [30] and *Bow-Tie Diagrams* [22]. These graphical formalisms address *static systems* where the order of event occurrences does not matter, and allow a direct representation of combinatorial structure functions as Boolean functions over basic events interpreted as propositions. Dynamic fault trees [24] extend fault trees to handle *dynamic systems* where event ordering matters, by adding logic gates where subtree ordering encodes temporal sequencing constraints. Dynamic systems are also traditionally modeled using Markov chains. In *non-repairable systems*, new events can only degrade the health of the system, which translates to monotony properties of structure functions. In *repairable* systems, a new event can improve the health of the system by prompting a repair action. Boolean logic-driven Markov Processes [13, 16, 32, 33] allow to address dynamic repairable systems.

Model-checking tools such as PRISM [35, 36] or UPPAAL-SMC [19, 17], supporting formalisms like Continuous-time Markov chains (CTMC) or Probabilistic Timed Automata (PTA), can be used for reliability analysis. Generalized Semi-Markov Processes (GSMP), which are strictly more expressive than CTMC and PTA, have also been quite successful for reliability analysis using Monte-Carlo [37, 23, 48] or bounded model-checking approaches [2]. Works such as [25] propose a superset of both GSMPs and PTAs and leverage either Monte-Carlo simulation or PRISM as back-end depending on the particular subset the model falls in.

In all of the above formalisms, system architecture, components, failure modes and failure propagation are not first class concepts, the concept of failure condition is implicit and cannot be disentangled from models and models are not composable. Moreover, design-space formalization is impossible with these formalisms, for their lack of generic modeling features and inability to express parametric system families.

The more recent *Model-Based Safety Analysis* (MBSA) approach [38] addresses these issues by adopting hierarchical modeling, failure modes, propagation rules and failure conditions as first-class concepts. A first collection of works proposes to annotate a functional design model with failure mode propagation rules: [20] proposes a safety extension for the well known AADL system design language; [31] extends a Simulink model with Boolean formulas modeling failure mode propagation conditions; in xSAP [12] a reference functional model is annotated with timed failure propagation information.

Extending a functional model with safety information is debatable, due to the fact that fault propagation can occur through non-functional paths in real systems, and that external non-functional factors also need to be modeled to conduct safety assessment. The computation of safety indicators requires to abstract away safety-irrelevant aspects of system behaviour to become tractable, and results in models that are qualitatively different from engineering models. Another line of works in MBSA addresses these issues by proposing languages dedicated to safety modeling. In particular, the Altarica family of languages [4, 42, 9] proposes a hierarchical modeling approach based on components and data-flow with a semantics based on Stochastic Guarded Transition Systems (SGTS). This framework is at least as expressive as GSMP and allows to model dynamic and repairable systems, with concurrency and real-time aspects, with deterministic or stochastic failure mode propagation rules, common-cause failure modeling with event synchronizations. The recent S2ML framework [8] uses concepts borrowed from object-oriented programming to improve model reuse and allow the creation of component libraries, and only offers a restricted form of parametricity.

Alpacas is a new incarnation of SGTS with hierarchical modeling and expressivity comparable to Altarica. However, Alpacas is tailored for design-space exploration by adding first-class support for generic modeling based on functional programming concepts such as higher-order parameters, typeclass polymorphism, etc. Design-space formalization, was only handled externally and informally in all previous approaches. In addition, Alpacas removes the strict boundary between safety models and analysis algorithms, opening the way to better design-space exploration methods.

2.3 Domain-specific languages

Domain-Specific Languages (DSL) are dedicated to the modeling and solving of particular classes of problems, and are generally not complete programming languages. *Standalone DSLs* are implemented by writing a standalone front-end (lexer, parser, type-checker, ...) and back-end (interpreter, compiler, solver, optimizer, ...). *Embedded DSLs* on the other hand are implemented within a host language [28], and exposed to the user through functional combinators or syntax extensions. Language embedding allows to reuse the host language syntax, type system, semantics, libraries, compilers and tools at the cost of slightly less freedom in the syntax definition of the DSL, and has become a very popular approach. A DSL embedding is *shallow* when DSL constructs are directly interpreted in the host language without any further analysis or code generation stages. The *Tagless Final* approach [34] is very popular for shallow DSL implementation: DSL operations are represented as a purely functional interface parameterized by a monadic higher-kinded effect type, which defines its semantics. In *deep embedding* approaches, evaluating the domain-specific program yields a term data structure representing the DSL program that is then analyzed and processed in multiple stages [44]. Deep embedding approaches based on free monads have been proposed, however both shallow and deeply embedded monadic approaches are hard to scale to large DSLs, are syntactically constrained by the monadic programming style, and require deep understanding of monads and higher-kinded types from the end user.

To implement ALPACAS, we opted for a non-monadic deep embedding technique, because the language is relatively rich and requires advanced static checks and preprocessing on the models before running simulations and analyses. The Scala language is known to offer very good support for deep embedding and staging, as demonstrated in multiple domains like hardware description with the Chisel language [5], Lightweight Multi-Stage numerical code optimization [44], full language virtualization [43], GPU acceleration of numerical code [49], event monitoring with automata [26], polymorphic linear algebra [45], etc. The newly released Scala 3 based on the Dependent Object Type calculus [3] offers even better support for deep embedding, with generalized algebraic data types, extension methods, infix methods, contextual abstraction mechanisms such as type-classes and automatic type-class derivation, and more importantly implicit function types [40], etc. Support for Multi-stage programming is also improved with the new inline-def macro system which, together with a new quoting and splicing system, provides efficient compile-time as well as run-time code generation.

3 Generic modeling needs and running example

In this section we illustrate MBSA concepts on a simple powertrain model, consisting of two batteries providing power to two electric engines. The failure condition is the loss of both engines. A battery component, shown in Figure 1, has two internal states Ok and Fail, an exponential failure delay distribution of parameter *lamB*. It produces a data-flow `power` representing the power failure mode, Ok in the Ok state and Fail in the Fail state. An engine component has two states Ok and Fail, an exponential failure delay distribution of parameter *lamE*. It produces a data-flow `thrust` representing the thrust failure mode, equal to its input `power` in the Ok state, and to Fail in the Fail state.

Components encapsulate states and guarded transitions behind a data-flow interface. Data-flow connections shown in Figure 2 model how failure modes propagate from batteries, to engines, to the failure condition observer through the system. Each engine's power input is connected to both batteries using an OR operator (produces Ok if one of the inputs is Ok, Fail otherwise). The engines' `thrust` outputs are connected to a failure condition observer

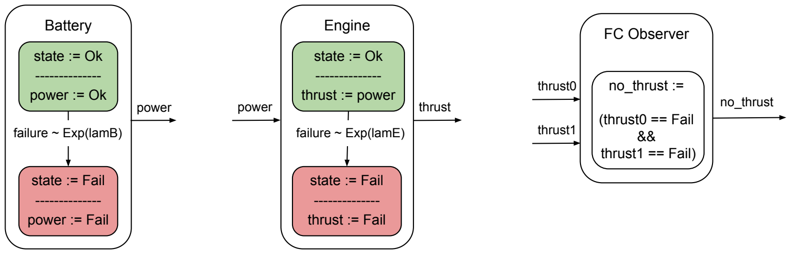

Figure 1 Powertrain example: battery, engine, failure condition observer components.

monitoring the loss of thrust on both engines. In the initial state shown in Figure 2(a), all components are in the Ok state and all power and thrust data-flows are Ok. The state in Figure 2(b) is reached after the failure of the first battery. Since the second battery is still Ok, engines still receive power and produce thrust and the failure condition is not triggered. The state in Figure 2(c) is reached after the failure of the second battery, which causes a loss of power for both engines and loss of thrust, despite the engines being in the Ok state. The failure condition is triggered as a result.

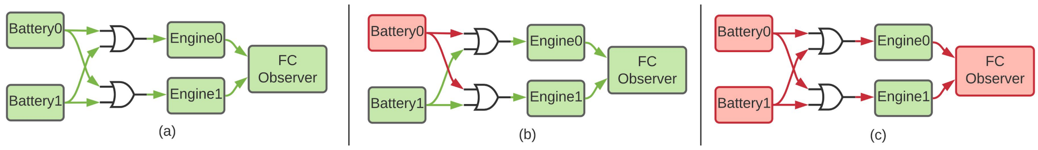

Figure 2 Powertrain example: state and flow updates after Battery0 and Battery1 failure events.

With Alpacas our goal is to formalize such a model in a generic way, where the number of engines and batteries are parameters, and where the topology of the power delivery connections between them is also a parameter of the model. This form of genericity affects the model's hierarchy as well as the topology of the data-flow network. We also want the concrete representation of failure states and failure modes of the engines and batteries to be parameters, as well as the delay distribution parameters of the corresponding events. By combining concepts from stochastic guarded transition systems and generic types, typeclass polymorphism and higher-order concepts from functional programming we can achieve this genericity. This genericity is the basis needed for genuine design-space formalization and exploration.

4 The Alpacas domain-specific language

This section presents the Alpacas DSL, the modeling workflow and the embedding techniques allowing the Scala syntax to be adapted to safety modeling needs. Section 4.1 to Section 4.5 introduce Alpacas constructs using the running example. Section 4.6 details the expressions language of Alpacas. Section 4.7 shows how we extended the Scala syntax for Alpacas.

Code examples with a green background show Alpacas code written by the end-user, and code examples with a red background show internal Alpacas implementation code. These examples are simplified compared to the actual library code, omitting the source mapping code which allows to track filenames, line numbers and Scala variable identifiers, handled using the `sourcecode` library. This implementation of Alpacas is written in Scala 3.0. Listing 1 presents the Alpacas encoding of the powertrain running example of section 3, which is later detailed in sections 4.1 to 4.5.

```
enum Failure derives Lifted {
  case Ok
  case Fail
}

import Failure.*

given Ord[Failure] with {
  def lt(x: Failure, y: Failure): Boolean = x == Ok && y == Fail
}

class Battery extends Component {
  val state   = State[Failure](init = Ok)
  val power   = OutFlow[Failure]
  val failure = Event(Exponential(1E-5))
  val repair  = Event(Dirac(5), weight = 1.0)
  assertions { power := state }
  transitions {
    When(failure) If state === Ok   Then {state := Fail}
    When(repair)  If state === Fail Then {state := Ok}
  }
}

class Engine extends Component {
  val state   = State[Failure](init = Ok)
  val thrust  = OutFlow[Failure]
  val power   = InFlow[Failure]
  val failure = Event(Exponential(1E-5), policy = Policy.Memory)
  val repair  = Event(Dirac(1))
  assertions {
    thrust := If (power === Ok && state === Ok) Then Ok Else Fail
  }
  transitions {
    When(failure) If(state === Ok && power === Ok) Then {state := Fail}
    When(repair)  If(state === Fail)               Then {state := Ok}
  }
}

type Batteries = Vector[Battery]; type Engines = Vector[Engine]
type Wiring    = (Batteries, Engines) => Assertions

class Powertrain(wiring: Wiring, n: Int) extends Component {
  val batteries = Subs(n)(Battery())
  val engines   = Subs(n)(Engine())
  val observer  = OutFlow[Boolean]
  val ccf       = Event(Exponential(1E-7))
  assertions {
    wiring(batteries, engines)
    observer := engines.map(_.thrust === Ok).reduce(_&&_)
  }
  transitions {
    Sync(ccf) With { batteries.map(_.failure.hard).reduce(_&_) }
  }
}

def one2one(b: Batteries, e: Engines): Assertions =
  e.map(_.power) := b.map(_.power)

def one2all(b: Batteries, e: Engines): Assertions =
  for (eng <- e) eng.power := b.map(_.power).reduce(_ min _)

val powertain121    = Powertrain(one2one, 2)
val powertrain12all = Powertrain(one2all, 2)
```

Listing 1 ALPACAS modeling of the powertrain example (cf Figure 1 for graphical view).

4.1 Lifting types, declaring components, state and flow variables

ALPACAS supports Scala's built-in *Boolean*, *Int* and *Double* types. Any Scala enumerated type can be lifted in the DSL and used to model component states and failure modes. Lines 1-4 of Listing 1 define a `Failure` enum with two values `Ok` and `Fail`, and lift it in ALPACAS space using the `derives Lifted` clause. The mechanism allowing this syntax will be detailed in Section 4.7.

It is possible to define ordering relations on user-defined types in order to use the DSL's relational operators $<$, $\leq$, $\geq$, $>$, min, max in guards and data-flow expressions. Orderings facilitate the definition of generic failure conditions or failure mode consolidation logic that only require to know if a failure mode is worse or better than another, without knowing exactly the individual failure modes. Lines 8-10 of Listing 1 define failure mode `Ok` to be strictly lesser than failure mode `Fail`.

ALPACAS allows to specify SGTS in a modular and composable way, and to derive a flat SGTS automatically. All user-defined safety components are represented as Scala classes extending an abstract `Component` class provided by the ALPACAS library. Components encapsulate state and flow variable declarations, event declarations, groups of transitions and flow assertions and have a strongly typed defined data-flow interface.

The model structure is captured using object-orientation (classes) and composition. Components can be instantiated inside other components using their constructors and the `Sub` statement. Vectors of sub-components are declared with the `Subs` statement where the size of the vector is provided as first argument (See lines 43-44 in Listing 1). The hierarchy of an ALPACAS model represents the system's static architecture.

Components contain either state variables declared by specifying their type and initial value with `State[Type](initial)`, or oriented flow variables declared by specifying their type and interface orientation with `OutFlow[Type]` or `InFlow[Type]`. In lines 25-27 of Listing 1, we define the variables for the `Engine` component: the state variable of type `Failure` and initial value `Ok` represents the intrinsic failure state of the component, the `power` input flow of type `Failure` represents the status of the power supply, and the `thrust` output flow represents the status of the thrust provided by the engine. Listing 2 shows how to declare vectors of variables with the keywords `States`, `InFlows` and `OutFlows`, which take the vector size as parameter.

```
class VectorExample extends Component {
  val state   = States[Failure](init = Ok)(4)
  val inputs  = InFlows[Failure](4)
  val outputs = OutFlows[Failure](4)
}
```

Listing 2 Vectors of variables.

4.2 Declaring flow assertions

Flow assertions define the flow variables in function of the state variables. Each component must define all its locally declared output flow variables, as well as all input flow variables of its sub-components. Line 31 of Listing 1 defines the `thrust` output of the `Engine` component to be `Ok` if the engine doesn't have an internal failure and receives nominal power supply. ALPACAS offers an overloaded flow definition operator := which works with equally sized vectors as left and right hand sides, as shown in line 57 of Listing 1. Functional iterators or for comprehensions can also be used to define vectors of flows point-wise, as shown in line 60 of Listing 1. A flow variable can be defined using any expression over flow or state variables as long as no cyclic flow dependency is introduced. Cyclic definitions are checked by the tool and reported to the user as hard errors (see Section 6.1).

4.3 Declaring transitions and synchronizations

Guarded transitions specify how the system state evolves over time. They are labeled by an event, and composed of a guard (a Boolean expression that must be true for the transition to be fired), and a set of state assertions (specifying how state variables are modified when the transition is fired).

Events represent random faults or deterministic system reactions and carry their delay distribution. Random faults are usually modeled using Exponential distributions, Weibull distributions, etc. Deterministic failure propagation or functional reactions of the system are modeled as events with Dirac distributions. When a transition is fireable, its firing delay is sampled from the distribution associated to its event (Dirac distributions produce a deterministic value). The default behaviour is to sample a new delay every time the transition becomes fireable, but it is also possible to store the delay when the transition stops being fireable and to use the stored delay value the next time it becomes fireable. This is called the *Memory* policy, it is useful to model components that wear out during their use. In line 28 of Listing 1, the failure event for engines is declared with a Memory policy, to model that if the engine is shut down because of a battery failure, when the battery is repaired the engine has the same remaining life as when it stopped being powered.

In order to support common cause modeling, ALPACAS offers event synchronization constructs, which express that two or more events can occur simultaneously as a consequence of another event named the *common cause*. The synchronized events can be either:

- *hard-synchronized*: all guards have to be true for the synchronized transition to be fired,
- *soft-synchronized*: at least one of the guards has to be satisfied for the synchronized transition to be fired. The state variables of soft-synchronized transitions are updated only if their guard was satisfied.

Line 52 of Listing 1 shows the hard-synchronization of the failures of two different batteries under a common cause failure event `ccf` (declared on line 46) that models a failure event affecting both engines at the same time (for instance a fire event, a lightning strike event, etc.). The `repair` event of the `Battery` component is declared with `Dirac(5)` delay distribution and `weight` parameter of 1.0 on line 16. The weight parameter is used to handle tie breaks between concurrent events. Here, following a `ccf` event, both batteries' `repair` events will be in concurrency. Tie breaks are achieved by selecting sampling a categorical distribution built from the from the weights of the concurrent events, here such that $p(\texttt{batteries(0).repair}) = p(\texttt{batteries(1).repair}) = \frac{1.0}{1.0+1.0} = 0.5$.

Line 28 of Listing 1 shows how to declare an Exponential distribution for the `failure` event of an engine, and line 29 a Dirac distribution for the functional `repair` event.

4.4 Specifying failure conditions

Any Boolean-valued data-flow of the model can be used as failure condition. For instance, the `observer` flow defined on line 49 of Listing 1 becomes false when the thrust of at least one engine is not `Ok`. Such definitions are usually placed in *observer components*, which are instantiated alongside the other components in the system. Several observers can exist in the system, however analyses take a single failure condition as parameter. Minimal sequences generation searches for event scenarios falsifying the condition. Unreliability analysis estimates the probability of this data-flow becoming false over some mission time T.

4.5 Parameters, type parameters, higher-order parameters

Component constructors can take parameters, allowing for instance to parameterize the number of sub-components or the number of state or flow variables of the component. Functional iterators (map, fold, reduce, ...) and vector assertions allow to define size-agnostic expressions, guards, assertions sets, etc.

Line 42 of Listing 1 declares the Powertrain Component, parameterized by the number of engines and batteries. Batteries and engines are declared as vectors of identical size on lines 43-44. Their data-flow connections are defined by a higher-order `wiring` parameter of type `Wiring`. The `Wiring` type, declared on line 40, is a function type taking `Batteries` and `Engines` vector inputs and producing an implicit function type *Assertions* (provided by the ALPACAS library) as output. The observer expression is defined as the conjunction of all engines providing thrust using the `reduce` iterator. Wiring schemes 1-to-1 and 1-to-all are defined respectively on lines 56-57 and 59-60. Two system variants with two engines and batteries and different wiring schemes are created using the `Powertrain` constructor on lines 62 and 63.

Type-class polymorphism allows to abstract over failure modes and to define generic flow aggregation logic, as shown in the voter example of Listing 3.

```
class Voter[A:Lifted:Ord](n: Int) extends Component {
  val inputs = InFlows[A](n)
  val output = OutFlow[A]
  assertions { output := inputs.reduce(_ max _) }
}
```

Listing 3 A generic voter component.

The example in Listing 4 shows how to use a trait and self-type annotation to define a reusable unit of behaviour. Using this trait we could for instance factor the failure logic between `Engine` and `Battery` components.

```
trait CanFail(lambda: Double) { self: Component =>
  val state = State[Failure](init = Ok)
  val fail  = Event(Exponential(lambda))
  transitions { When (fail) If (state === Ok) Then { state := Fail } }
}
class Engine extends Component with CanFail(lambda = 1E-7) { /* ... */ }
class Battery extends Component with CanFail(lambda = 1E-5) { /* ... */ }
```

Listing 4 Using traits to encapsulate reusable behaviour.

4.6 Abstract syntax for expressions

We use the initial algebra encoding approach for ALPACAS. Expressions are represented by abstract syntax trees defined inductively by a number of variants. Variants include flow variables, state variables, literal constants and constructors for all supported operations. The full abstract syntax is given below:

```
Expr ::= Const(value) | Svar(ident) | Fvar(ident) | Eq(Expr, Expr) |
         Ite(Expr, Expr, Expr) | Lt(Expr, Expr) | Un(Unop, Expr) |
         NumBin(NumBinop, Expr, Expr) | LogBin(LogBinop, Expr, Expr);

LogBinop ::= And | Or;  NumBinop ::= Add | Sub | Mult | Div;  Unop ::= Neg;
```

The following rules define well-typed expressions, where T is a generic type variable:

$$\frac{v \text{ of type } T}{\mathit{Const}(v) : T} \qquad \frac{s \text{ state variable of type } T}{\mathit{Svar}(s) : T} \qquad \frac{f \text{ flow variable of type } T}{\mathit{Fvar}(f) : T}$$

$$\frac{e_1 : T \quad e_2 : T}{\mathit{Eq}(e_1, e_2) : \mathit{Boolean}} \qquad \frac{c : \mathit{Boolean} \quad e_1 : T \quad e_2 : T}{\mathit{Ite}(c, e_1, e_2) : T}$$

The other constructs of the abstract syntax are defined only for some type-classes. We now present these type-classes and the corresponding typing rules.

Numeric is the type-class for numeric operations (addition, subtraction, multiplication and division), with typing rule:

$$\frac{e_1 : T \quad e_2 : T \quad \mathit{NumBinop} \in \mathit{Add}|\mathit{Sub}|\mathit{Mult}|\mathit{Div} \quad \mathit{Numeric}[T]}{\mathit{NumBin}(\mathit{NumBinop}, e_1, e_2) : T}$$

Logic is the type-class for Boolean operations (conjunction, disjunction and negation), with typing rules:

$$\frac{e_1 : T \quad e_2 : T \quad \mathit{LogBinop} \in \mathit{And}|\mathit{Or} \quad \mathit{Logic}[T]}{\mathit{LogBin}(\mathit{LogBinop}, e_1, e_2) : T} \qquad \frac{e : T \quad \mathit{Logic}[T]}{\mathit{Un}(\mathit{Neg}, e) : T}$$

Ord is the type-class of ordered types, with typing rule:

$$\frac{e_1 : T \quad e_2 : T \quad \mathit{Ord}[T]}{\mathit{Lt}(e_1, e_2) : \mathit{Boolean}}$$

The expression language and typing constraints are implemented in Scala 3 using the generalized algebraic datatype (GADT) shown in Listing 5. An implicit conversion for lifting Scala constants to expressions is also provided. ALPACAS expressions requiring a given type-class can only be constructed if an implicit type-class instance can be derived by the compiler for this type. This ensures that only well-typed expressions can be represented in the DSL. The type-checking of ALPACAS expressions is performed by the Scala compiler and type errors are highlighted in the IDE used for editing the models.

```scala
enum Expr[T] {
  case Const(value: T) extends Expr[T]

  case Svar(uid: StateId, init: T) extends Expr[T]

  case Fvar(uid: FlowId) extends Expr[T]

  case Eq(l: Expr[T], r: Expr[T]) extends Expr[Boolean]

  case Ite(c: Expr[Boolean], t: Expr[T], e: Expr[T]) extends Expr[T]

  case Lt(l: Expr[T], r: Expr[T])(using Ord[T]) extends Expr[Boolean]

  case NumBin(b: NumBinop, l: Expr[T], r: Expr[T])(using Numeric[T])
    extends Expr[T]

  case LogBin(b: LogBinop, l: Expr[T], r: Expr[T])(using Logic[T])
    extends Expr[T]

  case Un(u: LogUnop, e: Expr[T])(using Logic[T]) extends Expr[T]
}

given [T]: Conversion[T, Expr[T]] with {
  def apply(t:T): Expr[T] = Expr.Const(t)
}
```

Listing 5 Scala GADT for ALPACAS expressions.

4.7 Syntax extensions

As seen in the code examples of sections 4.1 to 4.3, ALPACAS provides constructs allowing to declare variables, assertions, transitions and expressions with a natural syntax. We use Scala 3's context abstraction capabilities to perform the required book-keeping of state and flow variables, events, assertions and transition declarations without adding clutter for the end-user. The code of Listing 6 presents the `State` variable constructor (`InFlow` and `OutFlow` variable constructors definitions are similar). This constructor takes an implicit argument of type `StateVarSet` from the surrounding `Component` instance, and creates a new `Expr.Svar` instance representing a state variable, adds it to the set of variables of the component, and returns it.

```
object State {
  def apply[T](init: T)(using svar: StateVarSet): Expr.Svar[T] =
    val res = new Expr.Svar[T](StateId(), init)
    svar += res
    res
}
```

Listing 6 State variable constructor.

To group assertions declarations in an `assertions` block, we use context functions and Odersky's *builder pattern* [40]. The builder pattern allows to build data structures with a declarative syntax, hiding side effects performed by builder methods. Multiple builder patterns can be nested by introducing intermediary builder methods.

Listing 7 shows the `assertions` builder method. It takes an implicit `ComponentBuilder` argument, used to perform all book-keeping declarations and definitions found inside a component, that is only available when in a surrounding `Component` instance. The field `flowAssertionBuilder` of the `builder` object is placed in the implicit scope of the `assertions` method to make it available to the `:=` assertion definition operator (itself defined as an extension method in the derived `Lifted` instance, see Listing 9). The `init` argument of the `assertions` method, with implicit function type `FlowAssertionBuilder ?=> Unit`, is provided by the user as a block containing flow assertions. Nesting the `FlowAssertionBuilder` inside the `ComponentBuilder` ensures that a compile-time error occurs when attempting to define flow assertions outside of an `assertions` builder method.

```
def assertions(init: FlowAssertionBuilder ?=> Unit)
              (using builder: ComponentBuilder) =
  given FlowAssertionBuilder = builder.flowAssertionBuilder
  init
```

Listing 7 assertions function for the builder pattern defining flow assertions.

The `transitions` builder uses three levels of nesting: the `transitions` builder method takes an implicit `ComponentBuilder`, which contains a `TransitionBuilder` object provided to the `When(e) If(g) Then { v := expr }` builder construct, which itself contains a `StateAssertionsBuilder` object provided to the `:=` state assertion definition operator.

`Lifted`, shown in Listing 8, is the type-class for types that can be lifted to ALPACAS expressions. It allows to compare expressions using the equality `===` operator. The `:=` overloaded operator allows to define state variables in transitions (cf. Section 4.3) and to define flow variables in assertions (cf. Section 4.2).

```
trait Lifted[T] {
  extension (x: Expr[T])
    def === (y: Expr[T]): Expr[Boolean]

  extension (x: Expr.Svar[T])
    def := (y: Expr[T]) (using a: StateAssertionBuilder): Unit

  extension (x: Expr.Fvar[T])
    def := (y: Expr[T]) (using a: FlowAssertionBuilder): Unit
}
```

Listing 8 Lifted type-class.

Automatic type-class derivation is used to relieve the user from manually defining the type-class instance (as shown in Section 4.1). For equality, the operator `===` lifts the comparison to an expression. The polymorphic variable assignment operators `:=` takes implicit `FlowAssertionBuilder` and `StateAssertionBuilder` and adds the corresponding assertion to it.

```
object Lifted {
  def derived[T]: Lifted[T] = new Lifted[T] {
    extension (x: Expr[T])
      def === (y: Expr[T]): Expr[Boolean] = Expr.Eq(x, y)

    extension (x: Expr.Svar[T])
      def := (y: Expr[T]) (using a: StateAssertionBuilder): Unit =
        a += StateAssertion(x, y)

    extension (x: Expr.Fvar[T])
      def := (y: Expr[T]) (using a: FlowAssertionBuilder): Unit =
        a += FlowAssertion(x, y)
  }
}
```

Listing 9 Derived instance of type-class Lifted.

Type-classes `Numeric`, `Logic` and `Ord` are implemented using generic traits defining the necessary operations on an abstract type. We have other type-classes defining the corresponding operations on ALPACAS Expressions as extension methods, and we use type-parametric givens to automatically derive instances of these type-classes.

Listing 10 shows the `Ord` syntax extensions for expressions. The user provides an instance of type-class `Ord` for lifted type `T` (see Section 4.1). The type-class `DSLord` provides syntax extensions for expressions of the `Ord` type, and the corresponding type-parametric given ensures `DSLord` instances can be derived from `Ord` instances.

```
trait Ord[T:Lifted] {
  def lt(x: T, y: T): Boolean
}

trait DSLord[T: Lifted] {
  extension (x: Expr[T])
    def <   (y: Expr[T]): Expr[Boolean]
    def >   (y: Expr[T]): Expr[Boolean] = !(x < y) && !(x === y)
    def <=  (y: Expr[T]): Expr[Boolean] = x < y || x === y
    def >=  (y: Expr[T]): Expr[Boolean] = !(x < y)
    def min (y: Expr[T]): Expr[T]       = If (x < y) Then x Else y
    def max (y: Expr[T]): Expr[T]       = If (x < y) Then y Else x
}

given [T:Lifted:Ord]: DSLord[T] with {
  extension (x: Expr[T])
    def < (y: Expr[T]): Expr[Boolean] = Expr.Lt(x, y)
}
```

Listing 10 Type-class mechanism for ordered types.

For conditional flow selection, we use functions and infix methods to produce `IfThenElse` expressions as presented in Listing 11. Due to Scala parsing rules, the parenthesis are mandatory around the conditional but optional around the branches

```
def If(c: Expr[Boolean]): Ift = Ift(c)

case class Ift(c: Expr[Boolean]){
  def Then[T] (t: Expr[T]): IfThent[T] = IfThent(c, t)
}

case class IfThent[T](c: Expr[Boolean], t: Expr[T]){
  def Else (e: Expr[T]): Expr[T] = Expr.Ite(c, t, e)
}
```

Listing 11 Implementation of conditional statements.

5 Stochastic guarded transition systems

The semantics of an ALPACAS model is given by a Stochastic Guarded Transition System (SGTS). Our version of SGTS is largely inspired from [42, 9]. This formalism allows to model dynamic, repairable and re-configurable systems. From [42, 9], we reuse the notions of state and flow variables, Restart and Memory transitions, event concurrency resolution mechanisms and event synchronization mechanisms. However, we only accept causal systems and we add the notion of Urgent events. Urgent events have priority over all other events.

5.1 Definitions

▶ **Definition 1** (Stochastic Guarded Transition System). *A* Stochastic Guarded Transition System *is a tuple:*

$$SGTS = \langle S, F, A_F, T, E\rangle \tag{1}$$

Where:

- *S is a vector of typed **state variables**. Each state variable has an initial value v_{init};*
- *F is a vector of typed **flow variables** propagating failure modes through the system;*
- *A_F is a set of **flow assertions** of the form $v := expr$, with $v \in F$ and $expr$ an expression over state and flow variables defining v at all times;*
- *T is a set of **guarded transitions** of the form $g \xrightarrow{e} A_S$ where:*
  - *e is an **event**, the **trigger** of the transition;*
  - *g is a Boolean expression over state and flow variables, the **guard** of the transition;*
  - *A_S is a set of **state assertions** of the form $v := expr$ with $v \in S$ and $expr$ an expression over sate and flow variables, describing updates applied to state variables when the transition is fired.*

  Transitions are of three different types, which condition the way they are scheduled in the system's behaviour:
  - ***Urgent** transitions have priority over all other transitions and are fired immediately after their guard becomes true, without delay.*
  - ***Restart** transitions have an associated firing delay distribution $dist(e)$ and an optional real-valued weight parameter $W(e)$. The firing delay is sampled from the distribution each time a state where the guard is true is reached.*

- ***Memory*** *transitions have an associated firing delay distribution* $dist(e)$ *and an optional real-valued weight parameter* $W(e)$*. The firing delay is sampled the first time the guard becomes true, and sampled again only after the transition is fired, when the guard becomes true again. When the guard becomes false, the current delay value is saved and restored the next time the guard becomes true.*

  The different transition types entail a partition of the set of transitions $T = T_U \cup T_R \cup T_M$*;*
- $E = E_U \cup E_R \cup E_M$ *is the set of events, partitioned by event type.*

Example 2 shows the flat SGTS encoding of the powertrain running example presented in Listing 1. The If-Then-Else expressions appearing in flow definitions are the result of rewriting the `min` operator in terms of core operators. The common cause `ccf` transition was rewritten using the rules presented in Section 5.3.

▶ **Example 2** (Powertrain SGTS, one2all wiring)**.**

$$
\begin{array}{lll}
S & = \{ & b_0.state(init := Ok),\ b_1.state(init := Ok),\ e_0.state(init := Ok),\ e_1.state(init := Ok)\ \} \\
F & = \{ & observer,\ b_0.power,\ b_1.power,\ e_0.power,\ e_0.thrust,\ e_1.power,\ e_1.thrust\ \} \\
A_f & = \{ & b_0.power := b_0.state,\ b_1.power := b_1.state, \\
 & & e_0.power := Ite(b_0.power < b_1.power,\ b_0.power,\ b_1.power), \\
 & & e_1.power := Ite(b_0.power < b_1.power,\ b_0.power,\ b_1.power), \\
 & & e_0.thrust := Ite(e_0.power = Ok \land e_0.state = Ok,\ Ok,\ Fail), \\
 & & e_1.thrust := Ite(e_1.power = Ok \land e_1.state = Ok,\ Ok,\ Fail), \\
 & & observer := e_0.thrust = Ok \land e_1.thrust = Ok\ \} \\
T_R & = \{ & b_0.state = Ok \land b_1.state = Ok \xrightarrow{ccf\sim Exp(1e-7)} \{b_0.state := Fail,\ b_1.state := Fail\}, \\
 & & b_0.state = Ok \xrightarrow{b_0.failure\sim Exp(1e-5)} \{b_0.state := Fail\}, \\
 & & b_0.state = Fail \xrightarrow{b_0.repair\sim Dirac(5), W=1} \{b_0.state := Ok\}, \\
 & & b_1.state = Ok \xrightarrow{b_1.failure\sim Exp(1e-5)} \{b_1.state := Fail\}, \\
 & & b_1.state = Fail \xrightarrow{b_1.repair\sim Dirac(5), W=1} \{b_1.state := Ok\}, \\
 & & e_0.state = Fail \xrightarrow{e_0.repair\sim Dirac(1)} \{e_0.state := Ok\}, \\
 & & e_1.state = Fail \xrightarrow{e_1.repair\sim Dirac(1)} \{e_1.state := Ok\}\ \} \\
T_M & = \{ & e_0.state = Ok \land e_0.power = Ok \xrightarrow{e_0.failure\sim Exp(1e-5)} \{e_0.state := Fail\}, \\
 & & e_1.state = Ok \land e_1.power = Ok \xrightarrow{e_1.failure\sim Exp(1e-5)} \{e_1.state := Fail\}\ \} \\
T_U & = \{\,\} &
\end{array}
$$

The expression language used in assertions (already detailed in section 4.6) supports Boolean expressions, integer and floating point numeric expressions as well as equality checks over user-defined enumerations types. We only consider well typed expressions and assertions. A *total valuation* α is a total function over $S \cup F$ assigning a value to each state variable and flow variable, that can be decomposed into a state variable valuation α_S and a flow variable valuation α_F. We assume a function *eval* which evaluates an expression in the context of a valuation α. In a given state, the valuation α_S is defined relative to the previous state's total valuation α, whereas the valuation α_F is defined relative to the current α_S.

We assume that A_F contains a definition for each flow variable. A flow variable v depends on a state or flow variable v' if v' occurs in the expression defining v in A_F. We only consider *causal* systems where flow dependency is acyclic, so that there exists a topological ordering of flow variables allowing to evaluate all flow assertions in a single pass to obtain a flow valuation $\alpha_F = propagate(\alpha_S)$. A transition $g \xrightarrow{e} A_S$ is *fireable* in the context of a total valuation α if and only if $eval(g, \alpha)$ is true. We say that a valuation α is *stable* if no urgent transition is fireable in α, and *unstable* otherwise. Urgent transitions allow to model immediate feedback loops while preserving causality: a cycle in data-flow definitions is broken by introducing a stateful element in the cycle and delaying flow propagation to the next logical step using urgent transitions. Restart transitions allow to model random failure events for memoryless components for which state history has no influence. Memory transitions allow to model random failures of components for which the state history has an influence.

5.2 Stochastic timed trace semantics

▶ **Definition 3** (Timed Trace). *The semantics of stochastic guarded transition system is given by timed traces of the form:*

$$TimedTrace = S_0 \xrightarrow{e_0} S_1 \cdots \xrightarrow{e_{i-1}} S_i \xrightarrow{e_i} S_{i+1} \ldots \xrightarrow{e_{n-1}} S_n \tag{2}$$

A trace is a sequence of states S_i connected by Restart or Memory transitions where

$$S = \langle \overline{\alpha}, \alpha, \Sigma, Mem, t \rangle$$

is such that:

- $\overline{\alpha}$ *is a (possibly unstable) valuation,*
- α *is a stable valuation,*
- $\Sigma : E_R \cup E_M \to \mathbb{R}^+ \cup \{+\infty\}$ *is an* event schedule *associating a firing delay to each restart and memory event,*
- $Mem : E_M \to \mathbb{R}^+$ *is an* event delay memory *associating a memorized delay to each memory event,*
- t *is a positive real value representing the timestamp of the state.*

Firing a transition $g \xrightarrow{e} A_S$ in the context of a stable or unstable valuation α (decomposed in α_S and α_F) yields a new valuation α' decomposed in α'_S and α'_F defined by:

$$\alpha'_S(v) = \begin{cases} eval(expr, \alpha) & \text{if } \{v := expr\} \in A_S \\ \alpha_S(v) & \text{otherwise} \end{cases} \tag{3}$$

$$\alpha'_F = propagate(\alpha'_S) \tag{4}$$

When in a state S_i, the Restart or Memory transition to fire is the one with the smallest delay in the event schedule, $e_i = \operatorname{argmin}(\Sigma_i)$. If several events have the same minimum delay value, the weight values of the concurrent events are used to break the tie. A categorical distribution is created such that $p(e) = \frac{W(e)}{\sum_{e \in \mathbf{argmin}(\Sigma_i)} W(e)}$, and the event e_i is sampled from this distribution.

The (possibly unstable) valuation $\overline{\alpha}_{i+1}$ is the result of firing the transition associated to event e_i in the stable valuation α_i.

The stable valuation α_{i+1} is determined by exploring all possible interleavings of fireable urgent transitions starting from $\overline{\alpha}_{i+1}$, transitively across unstable valuations. If all interleavings lead to the same stable valuation α_{i+1}, it is taken as the stable valuation for the successor state S_{i+1}, otherwise the trace is considered invalid.

- For each Restart event e, the schedule at state $i+1$ is defined depending on whether e is the event e_i that was fired in state i or not, and on its fireability in states i and $i+1$:

$e = e_i$	$\mathit{fireable}(e, \alpha_i)$	$\mathit{fireable}(e, \alpha_{i+1})$	$\Sigma_{i+1}(e)$
$\top$	$\top$	$\top$	$d \sim \mathit{dist}(e)$
$\top$	$\top$	$\bot$	$+\infty$
$\bot$	$\top$	$\top$	$\Sigma_i(e) - \Sigma_i(e_i)$
$\bot$	$\top$	$\bot$	$+\infty$
$\bot$	$\bot$	$\top$	$d \sim \mathit{dist}(e)$
$\bot$	$\bot$	$\bot$	$+\infty$

- For each Memory event e, the schedule and memory functions at state $i+1$ are defined depending on whether e is the event e_i that was fired in state i or not, on its fireability in states i and $i+1$, and on the value of its delay memory in state i:

$e = e_i$	$\mathit{fireable}(e, \alpha_i)$	$\mathit{fireable}(e, \alpha_{i+1})$	$\mathit{Mem}_{i+1}(e)$	$\Sigma_{i+1}(e)$
$\top$	$\top$	$\top$	$d \sim \mathit{dist}(e)$	$\mathit{Mem}_{i+1}(e)$
$\top$	$\top$	$\bot$	$d \sim \mathit{dist}(e)$	$+\infty$
$\bot$	$\top$	$\top$	$\Sigma_i(e) - \Sigma_i(e_i)$	$\mathit{Mem}_{i+1}(e)$
$\bot$	$\top$	$\bot$	$\Sigma_i(e) - \Sigma_i(e_i)$	$+\infty$
$\bot$	$\bot$	$\top$	$\mathit{Mem}_i(e)$	$\mathit{Mem}_{i+1}(e)$
$\bot$	$\bot$	$\bot$	$\mathit{Mem}_i(e)$	$+\infty$

- $t_{i+1} = t_i + \Sigma_i(e_i)$ (the time progresses by the fired event's delay value).

The initial state S_0 of a timed trace is defined by:

- $\overline{\alpha}_{S0}(v) = v_{init}$ for all state variables,
- $\overline{\alpha}_{F0}(v) = \mathit{propagate}(\overline{\alpha}_{S0}(v))$,
- α_0 is obtained by exploring all interleavings of Urgent events starting from $\overline{\alpha}_0$ as described above,
- For each Restart event e:

$$\Sigma_0(e) = \begin{cases} d \sim \mathit{dist}(e) & \text{if } \mathit{fireable}(e, \alpha_0) \\ +\infty & \text{otherwise} \end{cases}$$

- For each Memory event e:
  - $\mathit{Mem}_0(e) = d \sim \mathit{dist}(e)$,
  - $\Sigma_0(e) = \begin{cases} \mathit{Mem}_0(e) & \text{if } \mathit{fireable}(e, \alpha_0) \\ +\infty & \text{otherwise} \end{cases}$
- $t_0 = 0$

5.3 Event synchronizations

It is possible to define synchronizations of several Restart and Memory events (but not Urgent events) with another event called the *common cause event*. The common cause event can have its own delay distribution and weight parameter.

▶ **Definition 4** (Synchronization). *A synchronization has the form:*

$$(e \ : \ a_1.\mathit{hard} \ \& \ \cdots \ \& \ a_m.\mathit{hard} \ \& \ b_1.\mathit{soft} \ \& \ \cdots \ \& \ b_n.\mathit{soft}) \ \xrightarrow{g} \ A_S$$

Where

- *e is the common cause event,*
- $\{a_i.hard \mid 0 \leq i \leq m\}$ *are the* mandatory events *of the synchronization,*
- $\{b_i.soft \mid 0 \leq i \leq n\}$ *are the* optional events *of the synchronization,*
- *g is a (possibly true) guard,*
- *A_S is a (possibly empty) set of state assertions.*

The semantics of a synchronization is defined by translation to the core formalism. We assume that the transitions corresponding to synchronized events are already rewritten to standard transitions if they were synchronized transitions, so that we have a set of mandatory transitions of the form: $M = \{h_1 \to A_{s_1}, ..., h_l \to A_{s_l}\}$ and a set of optional transitions of the form: $O = \{j_1 \to B_{s_1}, ..., j_n \to B_{s_n}\}$.

We denote by *If g Then B_s* the set of state assertions B_s where each assertion $v := expr$ is rewritten to $v :=$ *If g Then e Else v*. The translation is defined as follows:

- **Case $l > 0$**: The synchronization rewrites to:

$$h_1 \,\&\&\, ... \,\&\&\, h_l \xrightarrow{e} A_{s_1} \cup ... \cup A_{s_l} \cup \mathit{If}\ j_1\ \mathit{Then}\ B_{s_1} \cup .. \cup \mathit{If}\ j_n\ \mathit{Then}\ B_{s_n}$$

- **Case $l = 0$ and $n > 1$**: The synchronization rewrites to:

$$j_1 \,||\, ... \,||\, j_l \xrightarrow{e} \mathit{If}\ j_1\ \mathit{Then}\ B_{s_1} \cup .. \cup \mathit{If}\ j_n\ \mathit{Then}\ B_{s_n}$$

- **Case $l = 0$ and $n = 1$**: The synchronization rewrites to:

$$\mathit{true} \xrightarrow{e} \mathit{If}\ j_1\ \mathit{Then}\ B_{s_1}$$

5.4 Instability, Zeno phenomena and other issues

The definitions given in the previous sections do not prohibit ill-conditioned systems where the following issues occur:

- multiple distinct stable valuations are reachable from a given unstable valuations,
- the system exhibits Zeno behaviour, i.e. can take an infinite number of transitions through unstable valuations, or through stable states or a combination of both in a finite amount of time,
- event concurrency situations which cannot be solved because of a missing weight parameter (which we handle as a modeling error from the user),
- systems with unwanted deadlock states due to synchronizations of transitions with incompatible guards, etc.
- runtime errors in expression evaluation such as arithmetic underflow/overflow, division by zero, etc.

Static analysis or model-checking algorithms allow to detect such issues ahead of time, however in this first version of Alpacas we detect such problems at run-time when exploring event sequences or simulating the system, leaving the more advanced method for future work. Detection is performed by monitoring diverging interleavings of urgent transitions; monitoring for cycles of unstable states; exiting in error if a threshold was exceeded on the number of fired events (including urgent events) without having time progress; exiting in error in case an event without weight parameter is involved in a concurrent race. We also offer an interactive step simulator that allows the user to test the model against their own expectations.

6 Alpacas algorithms

This section presents the main algorithms available in Alpacas allowing to process a model and compute its safety indicators: flattening, basic evaluation and step simulation, minimal cut sequence enumeration, stochastic simulation.

6.1 Translating a hierarchical model to a flat stochastic guarded transition system

Hierarchical models need to be translated to the underlying SGTS representation to be analyzed. Since the hierarchy is flattened in the process, this translation is called `flattening`.

The first part of the flattening is to traverse the structure recursively to collect all variables, assertions and transitions of the model. We store them in adequate structures referencing them by their unique identifiers. We also generate human-readable names for variables and events reflecting to their full path in the component hierarchy.

Then several checks are performed. We use the `cats` library's `Validated` type to accumulate errors of several parallel validation tasks. The first check is for flow definitions: we verify that each component actually defines exactly once all the flows it must define (its output flows and its sub-components' input flows). If it is not the case, we accumulate all errors corresponding to missing or redundant definitions (with variables names and line of declaration) and send back the errors to the user. The second check is for model causality: we verify that the flow dependency is not cyclic. To do this, we generate the graph representing the dependency relation between flow variables defined by flow definitions (we use the `scalagraph` library). The absence of cyclic definitions is verified if and only if every strongly connected component of the graph contains only one node and flow assertions do not create direct self-dependencies. We check this using `scalagraph`, and in case of failure produce an error describing all variables involved in every cyclic component of the dependency graph. If no error is found, we compute a topological ordering on the graph that allows to compute flow variable assignments in sequential order.

Finally, we rewrite synchronizations to standard transitions according to the definitions presented in Section 5.3. This is done thanks to a recursive function that we call on every transition. Every time a synchronization is found, we recursively flatten the synchronized events (that can themselves correspond to synchronizations).

6.2 Transition firing and state updates

The basis of all analyses that can be made on an SGTS is the representation of α_S and α_F valuations and how they are updated to reflect the firing of a transition, moving one step forward in the trace of a valid run of the SGTS.

As described in Section 5.2, firing a transition consists in computing the new state valuation according to the previous total valuation and to the state assertions of the fired transition, followed by computing the flow valuation according to the new state valuation and to all flow assertions in topological order, iterating this process as long as urgent transitions are possible, to finally reach a stable valuation or exit in error if divergent urgent behaviour is detected or Zeno behaviour is detected.

Another important basic function used in all algorithms is the computation of the list of fireable transitions. This is straightforward from the evaluation of all transitions guards in a given state.

These two building blocks allow us to provide an interactive step simulator. When in this mode, the values of variables and fireable transitions in the current assignment are displayed to the user who can manually choose the next transition to fire (instead of using the minimum delay rule of the timed trace semantics). The next state is then displayed (with an option for displaying only the state and flow variable delta with respect to the previous state), so on and so forth until the user stops the simulation. Thanks to the functional immutable data structures backing this simulation mode, the user can undo previous decisions at any point and backtrack in the simulation in order to explore another branch.

6.3 Qualitative indicators

The enumeration of minimal sequences requires to produce traces that lead to a state satisfying a failure condition. To avoid redundancies, only minimal failure scenarios according to a given partial ordering over sequences are considered in safety analysis. We support the most common ordering used in the safety literature, which is the subsequence relation. To generate all possible minimal sequences, we explore the set of possible failure sequences using a bounded breadth-first search algorithm, allowing to generate sequences that are minimal by construction: sequences of size n are naturally explored only after all sequences of smaller sizes are explored. We also avoid visiting extensions of sequences that are already known to satisfy the failure condition.

```
val queue = Queue((immutableInitialState(model), List[EventId]()))
var res: List[List[EventId]] = Nil
while (!queue.isEmpty)
  val (state, seq) = queue.dequeue()
  if (eval(failureCondition, state) && !res.exists(subSequence(_, seq)))
    res = seq::res
  else if (seq.size < maxSize)
      val ftrans = fireable(model, state)
      ftrans.foreach{t =>
        val newSeq = t.id::seq
        if (!res.exists(subSequence(_, seq)))
          val newstate = fire(state, t.id)
          queue.enqueue((newState, newSeq))
      }
res
```

Listing 12 Breadth-first search with online minimization for minimal sequences enumeration.

From the minimal cut sequences we can deduce the minimal cutsets by forgetting the order and eliminating redundancies. If the system is static, this operation doesn't remove any information (the minimal sequences correspond to all permutations of the minimal cutsets), but if it is dynamic, we possibly lose information about the dysfunctional behaviour of the system (the exact ordering of events required to trigger a failure condition), which however translates to safe pessimism for the analysis. Due to the combinatorial explosion of the exploration for large systems, very high order cutsets are often neglected in order to scale the computations on large models. Low order cutsets (up to order 3) are the direct target of regulations and hence have the strongest impact on design decisions, and are the largest contributors to unreliability. Nevertheless, the probability of unexplored scenarios can be soundly approximated by considering they all trigger the failure condition.

We give in Table 1 the output given by the tool for minimal cutsets of the example given in Listing 1. The failure condition is the loss of thrust for one or more engine, the results are as expected: the intrinsic failure of either one engine or the other trigger the failure condition, as does the loss of both batteries, either by the combination of their failure events,

Table 1 MCS for `powertrain12all`.

Order	Minimal cutset
1	ccf
1	engines(0).failure
1	engines(1).failure
2	batteries(0).failure, batteries(1).failure

or by a common cause failure triggering the simultaneous loss of both batteries (a single battery loss is tolerated thanks to the one-to-all wiring). More efficient SAT or SMT-based model-checking techniques can also be used for minimal cutset [21] or minimal sequence enumeration [14], with an explicit time model [1] or without. Our initial focus being on language expressivity, we leave this as future work.

6.4 Quantitative indicators

▶ **Definition 5** (Reliability, Unreliability)**.** *Let t_{fail} be the random variable describing the instant at which system failure occurs.* ***Reliability*** *for a mission time T is defined as the probability that the system failure does not occur in the interval $[0, T]$, knowing that the system is in perfect nominal condition at time 0.* ***Unreliability*** *is the complement of reliability.*

$$R(T) = p(t_{fail} > T), \quad U(T) = 1 - R(T)$$

The reliability of the system can be computed from minimal cutsets using a BDD-based algorithm [41]. We provide an implementation of this algorithm using the JavaBDD library. It relies on the user-specified delay distributions for events (this analysis is offered only if all distributions are specified), and is evaluated for a given mission time T. The computation yields an exact result if it is based on all cutsets for a static system, and becomes a safe under-approximation if the system is dynamic. The computation yields a possibly unsafe approximation for both static and dynamic systems if cutsets of high order are neglected. This BDD-based analysis cannot take dynamic repair or reconfiguration events into account.

Monte-Carlo simulation on the other hand allows to take into account the dynamic repair and reconfiguration of a system without approximation. The Alpacas stochastic simulator allows to sample finite traces of an SGTS and to compute safety indicators on the fly, by directly folding traces using a statistics aggregation function, without storing the traces. We provide aggregators for usual safety indicators such as (un)reliability, availability, mean time between failures, etc. The Monte-Carlo estimates converge in $\frac{1}{\sqrt{\#samples}}$ and high-confidence intervals can be computed based on the empirical sample mean and variance. The Alpacas simulator supports multi-core parallelism thanks to Scala's parallel collections library.

Table 2 gives a comparison of the runtimes and results of the Minimal Cutsets + BDD method vs the Monte-Carlo method for unreliability estimation. Results were obtained on a quad core MacBook Pro 13" 2019 with 16gigs of Ram. For mission times up to 10^3 time units, Minimal Cutsets + BDD and Monte-Carlo results are equal up to the third decimal. The difference on the remaining decimals can be attributed to the natural imprecision of Monte-Carlo methods. For longer mission times, the Monte-Carlo unreliability is lower than the MCS unreliability. This is due to the repairability of the system which is neglected by the Minimal Cutsets + BDD technique. The computation cost for an estimation of the reliability is significantly higher for the Monte-Carlo method, and it increases with the duration of mission time, which is not the case for the Minimal Cutsets + BDD method. However, the cost of preliminary computations needed for each analyzed architecture must be taken into account. Flattening is necessary for both analyses while the computation of Minimal Custsets and the structure function's BDD are necessary only for the Minimal Custsets + BDD algorithm. For large models, the BDD computation typically becomes the bottleneck.

Table 2 Runtimes (ms) per preprocessing phase, MCS+BDD vs Monte-Carlo (10^5 samples, 95% confidence interval) and runtimes (ms) for Unreliability of `powertrain12all`.

Preprocessing Phase	CPU time
Flattening	377
MCS	13
BDD	18

T	$U(T)$ MCS+BDD	CPU time	$U(T)$ Monte-Carlo	CPU time
10^2	0.0020	< 1	0.00213 ± 0.00003	271
10^3	0.0201	< 1	0.0209 ± 0.0003	266
10^4	0.189	< 1	0.181 ± 0.002	307
10^5	0.919	< 1	0.867 ± 0.001	456

Importance sampling or importance splitting algorithms [29, 15] are well known techniques for rare event estimation that can scale better and converge faster than unbiased Monte-Carlo. However, deriving meaningful importance functions (typically real-valued functions) in our discrete setting requires further research. Recent property-directed algorithms for probabilistic model checking [10] mixing symbolic and quantitative analysis for Markov Processes look very promising, but would need to be generalized to be applicable to Alpacas models (Alpacas models can be semi-Markov and even more general due to the Memory transitions).

7 Design-space exploration for an eVTOL thrust reallocation function

The main objective of this case study is to demonstrate that the Alpacas feature set makes it indeed well suited for safety modeling (including dysfunctional and functional behaviour) and design-space exploration for system architectures involving varying numbers of components, and alternative data-flow connections schemes. Another goal is to illustrate the kind of system design tradeoffs that can be analyzed through design-space exploration.

For this purpose, we chose to model a thrust system for a multi-rotor eVTOL able to tolerate any single fault while preserving safe hovering capability. It requires to compensate thrust loss while preserving thrust symmetry. The approach used for thrust compensation is described in [7]. It consists in shutting down the engine opposite to the failing engine to maintain symmetry with respect to all rotational axes, and to reallocate the missing lift on the remaining engines by increasing (trimming) their default thrust value.

The choice of architecture for this thrust function is not obvious, and requires automatic exploration. We must take into account the failure modes of all components involved: Batteries, Engines, Sensors, and CPUs executing the thrust reallocation logic. Thrust loss can be due to a intrinsic engine failure, or to a failure of the batteries powering the engine. It can also be due to a failure of a sensor triggering a spurious trim. The reallocation logic itself can also be lost due to CPU malfunction, or due to a battery failure, etc. From a cost/reliability trade-off perspective, a design using few engines requires high trim levels and high nominal engine thrust, and hence larger and more powerful engines and batteries, which comes at a cost. A design using more engines requires smaller nominal thrusts and trim levels in single failure cases, possibly cheaper engines, and could tolerate double failures. It has other downsides like wiring complexity and increased weight and it still requires high trim values in double failure scenarios, possibly quickly degrading the health of small engines.

We propose a parametric family of architectures allowing to implement the reconfiguration logic. In this study, we propose a parametric ALPACAS model capturing the design-space, and compute safety indicators for a number of configurations to identify design tradeoffs, and select the safest architecture(s).

Figure 3 shows one of the many possible architectures for the system (engine positions in the picture do not reflect their actual position in the aircraft): 6 batteries, 6 engines, one sensor per engine, dual computing units, dual power redundancy for all components, shared power sources for diagonally opposed engines and sensors, segregated power sources for axially opposed engines.

The design-space to explore is parameterized by the number of batteries, engines and sensors $n \in \{6, 8, 10\}$, by the battery failure rate λ_b, by the sensor failure rate $\lambda_s \in \{1\text{E-}5, 1\text{E-}10\}$, by the default sensor readout when it is not working properly (either optimistic or pessimistic, Boolean parameter *opt*). The engine failure rate is a piecewise constant function of the trim value: λ_0 when in [0%,10%], λ_1 when in [10%, 50%], λ_2 when in [50%,100%]. We model two computing units of failure rates $\lambda_c = 1\text{E-}10$. We model dual redundant power source for engines, sensors and one-to-all wiring for computing units. We consider two power source segregation cases (Boolean parameter *seg*): one where a sensor and its engine have the same power source, another where they use different sources. For $n = 6$, the reconfiguration logic doesn't cover double engine failures as this could yield a situation with only 2 engines functioning (2 are failed and 2 are shutdown) resulting in a loss of control and out of range trim values. For $n \in \{8, 10\}$, the logic does trigger a reconfiguration in case of a double engine failure.

The failure rates and mission time chosen for this study are not realistic. Their relative orders of magnitude were simply chosen to illustrate their influence on reliability, and give the reader an idea of the kind of design decisions that can be studied using ALPACAS models and algorithms.

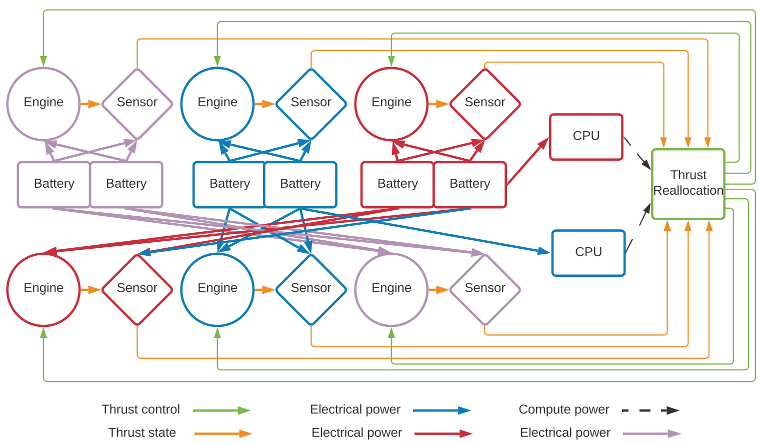

Figure 3 Conceptual diagram of the thrust reallocation system with 6 engines.

The design space exploration results are presented in Table 3. Results are obtained with 100 seconds of computation on a quad core MacBook Pro 13" 2019 with 16gigs of Ram. We use depth-first search for minimal sequences enumeration. We use Monte-Carlo with 100k simulations for unreliability estimation, to properly take the dynamic thrust reallocation behaviour into account. All configurations are immune to single failures (no minimal cutset of

Table 3 Design-space exploration results (mission time 10^3 time units).

n	λ_0	λ_1	λ_2	λ_b	λ_s	*opt*	*seg*	# order 1 mcs	# order 2 mcs	# order 3 mcs	$U(T)$ 95% conf. int.
6	1.0E-5	1.0E-4	2.0E-4	1.0E-5	1.0E-5	true	false	0	57	6	0.0252 ± 0.0003
6	1.0E-5	1.0E-4	2.0E-4	1.0E-5	1.0E-5	true	true	0	54	42	0.0253 ± 0.0003
6	1.0E-5	1.0E-4	2.0E-4	1.0E-5	1.0E-5	false	false	0	120	54	0.0481 ± 0.0006
6	1.0E-5	1.0E-4	2.0E-4	1.0E-5	1.0E-5	false	true	0	123	6	0.0480 ± 0.0006
6	1.0E-5	1.0E-4	2.0E-4	1.0E-5	1.0E-10	true	false	0	57	6	0.0248 ± 0.0003
6	1.0E-5	1.0E-4	2.0E-4	1.0E-5	1.0E-10	true	true	0	54	42	0.0234 ± 0.0003
6	1.0E-5	1.0E-4	2.0E-4	1.0E-5	1.0E-10	false	false	0	120	54	0.0247 ± 0.0003
6	1.0E-5	1.0E-4	2.0E-4	1.0E-5	1.0E-10	false	true	0	123	6	0.0251 ± 0.0003
8	2.0E-5	1.0E-4	2.0E-4	1.0E-5	1.0E-5	true	false	0	12	320	0.0102 ± 0.0001
8	2.0E-5	1.0E-4	2.0E-4	1.0E-5	1.0E-5	true	true	0	8	368	0.0106 ± 0.0001
8	2.0E-5	1.0E-4	2.0E-4	1.0E-5	1.0E-5	false	false	0	0	1568	0.0157 ± 0.0002
8	2.0E-5	1.0E-4	2.0E-4	1.0E-5	1.0E-5	false	true	0	4	1496	0.0153 ± 0.0002
8	2.0E-5	1.0E-4	2.0E-4	1.0E-5	1.0E-10	true	false	0	12	320	0.0094 ± 0.0001
8	2.0E-5	1.0E-4	2.0E-4	1.0E-5	1.0E-10	true	true	0	8	368	0.0094 ± 0.0001
8	2.0E-5	1.0E-4	2.0E-4	1.0E-5	1.0E-10	false	false	0	0	1568	0.0093 ± 0.0001
8	2.0E-5	1.0E-4	2.0E-4	1.0E-5	1.0E-10	false	true	0	4	1496	0.0102 ± 0.0001
10	2.5E-5	1.0E-4	2.0E-4	1.0E-5	1.0E-5	true	false	0	15	690	0.0260 ± 0.0003
10	2.5E-5	1.0E-4	2.0E-4	1.0E-5	1.0E-5	true	true	0	10	770	0.0264 ± 0.0003
10	2.5E-5	1.0E-4	2.0E-4	1.0E-5	1.0E-5	false	false	0	0	3490	0.0383 ± 0.0005
10	2.5E-5	1.0E-4	2.0E-4	1.0E-5	1.0E-5	false	true	0	5	3370	0.0381 ± 0.0005
10	2.5E-5	1.0E-4	2.0E-4	1.0E-5	1.0E-10	true	false	0	15	690	0.0236 ± 0.0003
10	2.5E-5	1.0E-4	2.0E-4	1.0E-5	1.0E-10	true	true	0	10	770	0.0247 ± 0.0003
10	2.5E-5	1.0E-4	2.0E-4	1.0E-5	1.0E-10	false	false	0	0	3490	0.0238 ± 0.0003
10	2.5E-5	1.0E-4	2.0E-4	1.0E-5	1.0E-10	false	true	0	5	3370	0.0253 ± 0.0003

order 1). Configurations with 8 and 10 engines can tolerate double failures using pessimistic sensor defaults and non-segregated power wirings. Using pessimistic sensor defaults leads to an explosion of the number of minimal cutsets of order 3, which can increase unreliability if sensors are not sufficiently reliable. Indeed, a failing pessimistic sensor causes a spurious thrust reallocation, which leads to a trimming regime where engines fail more often. This results in a higher unreliability for the configurations that tolerate double failures. This tradeoff can be solved by increasing sensor reliability but this is to balance with cost aspects.

Listing 13 shows the ALPACAS code which generates the design-space of the system and selects the configuration without MCS of order 1 and with the lowest unreliability. The results can be further processed using the full Scala language, opening the door to design optimization taking into account other aspects such as the cost of the components, etc.

```
case class EngParams(nEng: Int, lam0: Double, lam1: Double, lam2: Double)

class ThrustRealloc(
  val engineParams: EngParams,
  val lambdaSensor: Double,
  val optimisticSensor: Boolean,
  val wiring : Wiring,
) extends Component { /* Model declaration */ }

val systems = for {
  eps      <- List(
    EngParams(6, 1E-5, 1E-4, 2E-4),
    EngParams(8, 2E-5, 1E-4, 2E-4),
    EngParams(10, 2.5E-5, 1E-4, 2E-4)
  )
  lamSens <- List(1E-5, 1E-10)
  optSens <- List(true, false)
  wiring  <- List(stdWiring(eps.nEng), segWiring(eps.nEng))
} yield ThrustRealloc(eps, lamSens, optSens, wiring)

var minUR = Double.PositiveInfinity
var bestSystem: Option[GenericPowertrain] = None
```

```
for {
  system <- systems
  model  <- stochasticCheck(system)
  mcs    <- minimalCutSetsBFS(model, system.observer.isOk, 3)
  urRes  <- unreliability(model, system.observer.isOk, 1E3, nbSimus, 8)
  (_, urmax) = urRes
} {
  if (mcs.forall(_.events.size > 1) && urmax < minUR) then
    bestSystem = Some(system)
    minUR = urmax
}
```

Listing 13 Design-space exploration example.

This study confirms that ALPACAS is adapted to safety modeling of parametric families of architectures, and allows to compute safety indicators on the formalized design-space allowing to identify design tradeoffs and possibly to determine the optimal architecture with regard to a chosen metric (which might include other parameters than safety indicators, like cost).

8 Conclusion and Future Work

In this paper we presented ALPACAS, a domain-specific language for system safety modeling and analysis. Using stochastic guarded transition systems as underlying formalism, it allows to model a large class of dynamic and re-configurable systems. It extends the state of the art in model-based safety assessment by bringing many cutting edge features from Scala 3 for generic programming thanks to a deep embedding. Parametric polymorphism, type-class polymorphism, higher-order parameters, higher-kinded types, etc. open the way to more efficient modeling and design-space formalization and exploration for safety critical systems. The ALPACAS feature set was tested on a representative case study modeling a family of architectures for a thrust reallocation function for electric Vertical Takeoff and Landing aircraft. The scope of applications of ALPACAS is not limited to aerospace systems and can benefit other domains such as automotive, railway, etc. which have similar safety processes [39, 46]. ALPACAS is available under an academic open-source license on this repository `https://gitlab.com/maximebuyse/alpacas`.

The future work planned for ALPACAS is the following. First, we will study how Scala 3's new macro system can improve the Monte-Carlo simulation performance, by inlining and specializing assertion, guards and transition evaluation functions, removing boxing as much as possible and distributing simulations on several computing cores. Second, we would like to connect this safety-oriented framework to existing Scala frameworks for temporal logic property monitoring such as DejaVu [27] or TraceContract [6]. This would allow to validate temporal logic properties on complex re-configurable system before deploying the temporal logic monitors for runtime safety assurance, and to derive process and reliability requirements for various autonomy functions. This would allow to monitor divergence between system models and actual system behaviour, and to trigger model updates to bridge the modeling gap. Third, we will study the connection of ALPACAS to reinforcement learning frameworks, in order to study the synthesis of optimal policies for reconfiguration, repair and maintenance of complex critical systems.

References

1. Alexandre Albore, Silvano Dal Zilio, Guillaume Infantes, Christel Seguin, and Pierre Virelizier. A model-checking approach to analyse temporal failure propagation with altarica. In Marco Bozzano and Yiannis Papadopoulos, editors, *Model-Based Safety and Assessment*, pages 147–162, Cham, 2017. Springer International Publishing.
2. R. Alur and M. Bernadsky. Bounded model checking for GSMP models of stochastic real-time systems. In J.P. Hespanha and A. Tiwari, editors, *Hybrid Systems: Computation and Control, 9th International Workshop, HSCC 2006, Santa Barbara, CA, USA, March 29-31, 2006, Proceedings*, volume 3927 of *Lecture Notes in Computer Science*, pages 19–33. Springer, 2006. doi:10.1007/11730637_5.
3. Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. The essence of dependent object types. In Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella, editors, *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, volume 9600 of *Lecture Notes in Computer Science*, pages 249–272. Springer, 2016. doi:10.1007/978-3-319-30936-1_14.
4. André Arnold, Gérald Point, Alain Griffault, and Antoine Rauzy. The altarica formalism for describing concurrent systems. *Fundam. Informaticae*, 40(2-3):109–124, 1999. doi:10.3233/FI-1999-402302.
5. Jonathan Bachrach, Huy Vo, Brian C. Richards, Yunsup Lee, Andrew Waterman, Rimas Avizienis, John Wawrzynek, and Krste Asanovic. Chisel: constructing hardware in a scala embedded language. In Patrick Groeneveld, Donatella Sciuto, and Soha Hassoun, editors, *The 49th Annual Design Automation Conference 2012, DAC '12, San Francisco, CA, USA, June 3-7, 2012*, pages 1216–1225. ACM, 2012. doi:10.1145/2228360.2228584.
6. Howard Barringer and Klaus Havelund. Tracecontract: A scala DSL for trace analysis. In Michael J. Butler and Wolfram Schulte, editors, *FM 2011: Formal Methods - 17th International Symposium on Formal Methods, Limerick, Ireland, June 20-24, 2011. Proceedings*, volume 6664 of *Lecture Notes in Computer Science*, pages 57–72. Springer, 2011. doi:10.1007/978-3-642-21437-0_7.
7. Pierre-Marie Basset, Binh Dang Vu, Philippe Beaumier, Gabriel Reboul, and Biel Ortun. Models and methods at onera for the presizing of evtol hybrid aircraft including analysis of failure scenarios. In *AHS Forum 2018, May 2018, PHOENIX, United States*, 2018.
8. Michel Batteux, Tatiana Prosvirnova, and Antoine Rauzy. System Structure Modeling Language (S2ML). preprint, 2015. URL: https://hal.archives-ouvertes.fr/hal-01234903.
9. Michel Batteux, Tatiana Prosvirnova, Antoine Rauzy, and Leïla Kloul. The altarica 3.0 project for model-based safety assessment. In *11th IEEE International Conference on Industrial Informatics, INDIN 2013, Bochum, Germany, July 29-31, 2013*, pages 741–746. IEEE, 2013. doi:10.1109/INDIN.2013.6622976.
10. Kevin Batz, Sebastian Junges, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Philipp Schröer. Pric3: Property directed reachability for mdps. In Shuvendu K. Lahiri and Chao Wang, editors, *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II*, volume 12225 of *Lecture Notes in Computer Science*, pages 512–538. Springer, 2020. doi:10.1007/978-3-030-53291-8_27.
11. Pierre Bieber, Remi Delmas, and Christel Seguin. Dalculus - theory and tool for development assurance level allocation. In Francesco Flammini, Sandro Bologna, and Valeria Vittorini, editors, *Computer Safety, Reliability, and Security - 30th International Conference, SAFECOMP 2011, Naples, Italy, September 19-22, 2011. Proceedings*, volume 6894 of *Lecture Notes in Computer Science*, pages 43–56. Springer, 2011. doi:10.1007/978-3-642-24270-0_4.
12. Benjamin Bittner, Marco Bozzano, Roberto Cavada, Alessandro Cimatti, Marco Gario, Alberto Griggio, Cristian Mattarei, Andrea Micheli, and Gianni Zampedri. The xsap safety analysis platform. In Marsha Chechik and Jean-François Raskin, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as*

Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings, volume 9636 of *Lecture Notes in Computer Science*, pages 533–539. Springer, 2016. doi:10.1007/978-3-662-49674-9_31.

13 Marc Bouissou and Jean-Louis Bon. A new formalism that combines advantages of fault-trees and markov models: Boolean logic driven markov processes. *Reliab. Eng. Syst. Saf.*, 82(2):149–163, 2003. doi:10.1016/S0951-8320(03)00143-1.

14 Marco Bozzano, Alessandro Cimatti, Alberto Griggio, and Cristian Mattarei. Efficient anytime techniques for model-based safety analysis. In Daniel Kroening and Corina S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, volume 9206 of *Lecture Notes in Computer Science*, pages 603–621. Springer, 2015. doi:10.1007/978-3-319-21690-4_41.

15 Carlos E. Budde, Pedro R. D'Argenio, and Arnd Hartmanns. Automated compositional importance splitting. *Sci. Comput. Program.*, 174:90–108, 2019. doi:10.1016/j.scico.2019.01.006.

16 Pierre-Yves Chaux, Jean-Marc Roussel, Jean-Jacques Lesage, Gilles Deleuze, and Marc Bouissou. Systematic extraction of minimal cut sequences from a BDMP model. In *21st European Safety & Reliability Conference (ESREL 12), Jun 2012, Helsinki, Finland*, 2012.

17 Shengxin Dai, Mei Hong, and Bing Guo. A comparative study of reliability-ignorant and reliability-aware energy management schemes using UPPAAL-SMC. *Sci. Program.*, 2017:2621089:1–2621089:12, 2017. doi:10.1155/2017/2621089.

18 Patrick R. Darmstadt, Ralph Catanese, Allan Beiderman, Fernando Dones, Ephraim Chen, Mihir P. Mistry, Brian Babie, Mary Beckman, , and Robin Preator. Hazards analysis and failure modes and effects criticality analysis (fmeca) of four concept vehicle propulsion systems. Technical report, NASA/Boeing, 2019. URL: https://hummingbird.arc.nasa.gov/Publications/files/CR-2019-220217.pdf.

19 Alexandre David, Kim G Larsen, Axel Legay, Marius Mikučionis, and Danny Bøgsted Poulsen. Uppaal smc tutorial. *International Journal on Software Tools for Technology Transfer*, 17(4):397–415, 2015.

20 Julien Delange and Peter H. Feiler. Architecture fault modeling with the AADL error-model annex. In *40th EUROMICRO Conference on Software Engineering and Advanced Applications, EUROMICRO-SEAA 2014, Verona, Italy, August 27-29, 2014*, pages 361–368. IEEE Computer Society, 2014. doi:10.1109/SEAA.2014.20.

21 Kevin Delmas, Rémi Delmas, and Claire Pagetti. Smt-based architecture modelling for safety assessment. In *2017 12th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 1–8. IEEE, 2017.

22 Ewen Denney, Ganesh Pai, and Josef Pohl. Advocate: An assurance case automation toolset. In Frank Ortmeier and Peter Daniel, editors, *Computer Safety, Reliability, and Security - SAFECOMP 2012 Workshops: Sassur, ASCoMS, DESEC4LCCI, ERCIM/EWICS, IWDE, Magdeburg, Germany, September 25-28, 2012. Proceedings*, volume 7613 of *Lecture Notes in Computer Science*, pages 8–21. Springer, 2012. doi:10.1007/978-3-642-33675-1_2.

23 T. English and R. Heydor. Monte carlo simulation of markov, semi-markov, and generalized semi-markov processes in probabilistic risk assessment, final report. Nasa summer faculty fellowship program 2004, NASA, August 2005.

24 Majdi Ghadhab, Sebastian Junges, Joost-Pieter Katoen, Matthias Kuntz, and Matthias Volk. Safety analysis for vehicle guidance systems with dynamic fault trees. *Reliab. Eng. Syst. Saf.*, 186:37–50, 2019. doi:10.1016/j.ress.2019.02.005.

25 A. Hartmanns. MODEST - A unified language for quantitative models. In *Proceeding of the 2012 Forum on Specification and Design Languages, Vienna, Austria, September 18-20, 2012*, pages 44–51. IEEE, 2012. URL: http://ieeexplore.ieee.org/document/6336982/.

26 Klaus Havelund and Rajeev Joshi. Modeling with scala. In *International Symposium on Leveraging Applications of Formal Methods*, pages 184–205. Springer, 2018.

27 Klaus Havelund, Doron Peled, and Dogan Ulus. Dejavu: A monitoring tool for first-order temporal logic. In *3rd Workshop on Monitoring and Testing of Cyber-Physical Systems, MT@CPSWeek 2018, Porto, Portugal, April 10, 2018*, pages 12–13. IEEE, 2018. doi:10.1109/MT-CPS.2018.00013.

28 Paul Hudak. Building domain-specific embedded languages. *Acm computing surveys (csur)*, 28(4es):196–es, 1996.

29 Cyrille Jégourel, Axel Legay, and Sean Sedwards. Importance splitting for statistical model checking rare properties. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 576–591. Springer, 2013. doi:10.1007/978-3-642-39799-8_38.

30 Sohag Kabir. An overview of fault tree analysis and its application in model based dependability analysis. *Expert Systems with Applications*, 77:114–135, 2017.

31 Sohag Kabir, Yiannis Papadopoulos, Martin Walker, David Parker, Jose Ignacio Aizpurua, Jörg Lampe, and Erich Rüde. A model-based extension to hip-hops for dynamic fault propagation studies. In Marco Bozzano and Yiannis Papadopoulos, editors, *Model-Based Safety and Assessment - 5th International Symposium, IMBSA 2017, Trento, Italy, September 11-13, 2017, Proceedings*, volume 10437 of *Lecture Notes in Computer Science*, pages 163–178. Springer, 2017. doi:10.1007/978-3-319-64119-5_11.

32 Shahid Khan, Joost-Pieter Katoen, and Marc Bouissou. A compositional semantics for repairable bdmps. In António Casimiro, Frank Ortmeier, Friedemann Bitsch, and Pedro Ferreira, editors, *Computer Safety, Reliability, and Security - 39th International Conference, SAFECOMP 2020, Lisbon, Portugal, September 16-18, 2020, Proceedings*, volume 12234 of *Lecture Notes in Computer Science*, pages 82–98. Springer, 2020. doi:10.1007/978-3-030-54549-9_6.

33 Shahid Khan, Joost-Pieter Katoen, and Marc Bouissou. Explaining boolean-logic driven markov processes using gspns. In *16th European Dependable Computing Conference, EDCC 2020, Munich, Germany, September 7-10, 2020*, pages 119–126. IEEE, 2020. doi:10.1109/EDCC51268.2020.00028.

34 Oleg Kiselyov. Typed tagless final interpreters. In *Generic and Indexed Programming*, pages 130–174. Springer, 2012.

35 Marta Z. Kwiatkowska, Gethin Norman, and David Parker. PRISM: probabilistic model checking for performance and reliability analysis. *SIGMETRICS Perform. Evaluation Rev.*, 36(4):40–45, 2009. doi:10.1145/1530873.1530882.

36 Marta Z. Kwiatkowska, Gethin Norman, and David Parker. PRISM 4.0: Verification of probabilistic real-time systems. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 585–591. Springer, 2011. doi:10.1007/978-3-642-22110-1_47.

37 N. Limnios and G. Oprişan. *Semi-Markov Processes and Reliability*. Number 1 in Statistics for Industry and Technology. Birkhäuser, Boston, MA, 2001. doi:10.1007/978-1-4612-0161-8.

38 O. Lisagor, T. Kelly, and R. Niu. Model-based safety assessment: Review of the discipline and its challenges. In *The Proceedings of 2011 9th International Conference on Reliability, Maintainability and Safety*, pages 625–632, 2011. doi:10.1109/ICRMS.2011.5979344.

39 Joseph Machrouh, Jean-Paul Blanquart, Philippe Baufreton, Jean-Louis Boulanger, Hervé Delseny, Jean Gassino, Gerard Ladier, Emmanuel Ledinot, Michel Leeman, Jean-Marc Astruc, Philippe Quéré, Bertrand Ricque, and Gilles Deleuze. Cross domain comparison of System Assurance. In *Embedded Real Time Software and Systems (ERTS2012)*, Toulouse, France, 2012. URL: https://hal.archives-ouvertes.fr/hal-02170444.

40 Martin Odersky, Olivier Blanvillain, Fengyun Liu, Aggelos Biboudis, Heather Miller, and Sandro Stucki. Simplicitly: foundations and applications of implicit function types. *Proc. ACM Program. Lang.*, 2(POPL):42:1–42:29, 2018. doi:10.1145/3158130.

41 Antoine Rauzy. Binary decision diagrams for reliability studies. In *Handbook of performability engineering*, pages 381–396. Springer, 2008.

42 Antoine B Rauzy. Guarded transition systems: a new states/events formalism for reliability studies. *Proceedings of the Institution of Mechanical Engineers, Part O: Journal of Risk and Reliability*, 222(4):495–505, 2008.

43 Tiark Rompf, Nada Amin, Adriaan Moors, Philipp Haller, and Martin Odersky. Scala-virtualized: linguistic reuse for deep embeddings. *High. Order Symb. Comput.*, 25(1):165–207, 2012. doi:10.1007/s10990-013-9096-9.

44 Tiark Rompf and Martin Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls. *Commun. ACM*, 55(6):121–130, 2012. doi:10.1145/2184319.2184345.

45 Amir Shaikhha and Lionel Parreaux. Finally, a polymorphic linear algebra language (pearl). In Alastair F. Donaldson, editor, *33rd European Conference on Object-Oriented Programming, ECOOP 2019, July 15-19, 2019, London, United Kingdom*, volume 134 of *LIPIcs*, pages 25:1–25:29. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.ECOOP.2019.25.

46 Ioannis Sorokos, Luis P. Azevedo, Yiannis Papadopoulos, Martin Walker, and David J. Parker. Comparing automatic allocation of safety integrity levels in the aerospace and automotive domains. *IFAC - PapersOnLine*, 49(3):184–190, 2016. 14th IFAC Symposium on Control in Transportation Systems 2016. doi:10.1016/j.ifacol.2016.07.031.

47 Alain Villemeur. *Reliability, availability, maintainability and safety assessment, assessment, hardware, software and human factors*, volume 2. Wiley, 1992.

48 Håkan L. S. Younes and Reid G. Simmons. Solving generalized semi-markov decision processes using continuous phase-type distributions. In Deborah L. McGuinness and George Ferguson, editors, *Proceedings of the Nineteenth National Conference on Artificial Intelligence, Sixteenth Conference on Innovative Applications of Artificial Intelligence, July 25-29, 2004, San Jose, California, USA*, pages 742–748. AAAI Press / The MIT Press, 2004. URL: http://www.aaai.org/Library/AAAI/2004/aaai04-117.php.

49 Tian Zhao and Xiaobing Huang. Design and implementation of deepdsl: A DSL for deep learning. *Comput. Lang. Syst. Struct.*, 54:39–70, 2018. doi:10.1016/j.cl.2018.04.004.