UAS procedures model with system architecture for safety analysis

Charles Mathou, Kevin Delmas, Pierre de Saqui-Sannes, Jean-Charles Chaudemar

UAS procedures model with system architecture for safety analysis

Charles Mathou
Fédération ENAC ISAE-SUPAERO ONERA
Université de Toulouse, France
charles.mathou@isae-supaero.fr

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

Pierre de Saqui-Sannes
Fédération ENAC ISAE-SUPAERO ONERA
Université de Toulouse, France
pdss@isae-supaero.fr

Jean-Charles Chaudemar
Fédération ENAC ISAE-SUPAERO ONERA
Université de Toulouse, France
jean-charles.chaudemar@isae-supaero.fr

***Abstract*—As the number of unmanned aerial systems (UAS) keeps increasing, so do the safety risks they pose. One way of maintaining an acceptable risk level is that operational procedures are adequately designed and proven. Model-based approaches involve modeling procedures as a sequence of tasks with inputs and outputs. These tasks abstract away the complexity of the subsystem or actor who performs them. However, UAS procedures typically involve multiple actors and subsystems, each of which contributes to the risk of the operation. Accounting for these heterogeneous risk contributors allows new failure propagation paths to be revealed, understood and patched, leading to increased safety. In order to achieve this, we propose a methodology to connect the safety models of such contributors to our previous procedure models. We discuss and illustrate this methodology on a medium-sized fixed-wing UAV. We connect our procedure models to the UAV's functional architecture model, and use them to generate minimal sequences leading to a crash of the UAV. New sequences illustrating the contribution of the UAV's architecture are revealed that did not appear in our previous work on UAS procedures. This provides an opportunity to explore the contribution of the system's architecture to its overall safety through the procedures.**

***Index Terms*—AltaRica, MBSA, UAS, Procedure**

I. Introduction

Unmanned aerial vehicles (UAVs) usually fly in cooperation with a ground station and a supervising pilot at a minimum. This core set of actors and systems is called an unmanned aerial system (UAS). Typically, UAS can also interact with external actors such as air traffic controllers, GPS networks or other UAVs. Each of these actors, and each of an UAS's subsystems are exposed to various failures which can impact the operation as a whole. The failures of a human pilot are studied in the human factors domain. A UAS's autopilot software failures stem from the embedded software domain. Failures in the communications link are best studied from an electromagnetic perspective. As such, the risk contributors to a UAS's operation are multiple and diverse in nature. Taking into account such risk contributors allows for a safer system as safety analyses can reveal new failure propagation paths.

In order to do so, we chose to approach the problem of failure propagation through a UAS's procedures. These procedures can describe both maintenance and operation. Operational procedures describe step by step actions to be taken in various cases ranging from routine or nominal situations to in-flight emergencies. As such they have to cover all of the actors, systems and subsystems and their interactions in the various scenarios they cover. The variety of actors and subsystems they involve, and the necessary interactions between those, represent a good opportunity to explore the mechanisms of failure propagation.

Our aim is thus to leverage our previous work on flight procedures [1] to propose a methodology allowing to account for different risk contributors to the UAS's safety, using procedure models as a fulcrum. Our objectives is thus to connect a procedures model to a model of the system whose procedure are modeled. Once the models are connected, we intend to leverage their formal, safety-oriented paradigm to automatically compute minimal sequences of events leading to undesired states such as a crash of the UAS. Doing so would allow us to refine our understanding of how system-level failure events propagate and impact a UAS operation using the procedure viewpoint.

We use AltaRica to model the procedures and the UAS functions and hardware. AltaRica tools support Model-Based Safety Analysis (MBSA) [2] such as the computation of the smallest sequences of failures leading to undesired events (called minimal sequences, MSQ). This enables us to assess the impact of specific failures on the UAV and its operation from a global perspective.

The paper is organized as follows: Section II surveys related work. Section III provides background on procedures. We present our use-case in section IV, and describe types of connections in section V. We present our methodology in section VI. We discuss our results in section VII.

II. Related work

A. Modeling failure propagation

The STAMP/STPA hazard analysis technique proposed by [3] is a deductive approach that considers safety in terms of control rather than failures. Instead of basing the analysis solely on component failures, they consider controls themselves as the source of hazard. To that end they develop a methodology that relies on the identification of potentially unsafe control actions based on a predefined safety control structure. This method is however not designed equipped to provide easy-to-automate, formal safety analyses.

The functional resonance analysis method (FRAM) [4] is a theoretic method designed to perform accident analysis, but has since been modified to also allow prospective risk assessments. It allows for the modeling of complex socio-technical systems through a set of interconnected vertices representing system functions and a set of their inputs, outputs, resources, controls, preconditions and time constraints. However, the specification of the variability of these functions based on their various inputs still requires further work [5]. Implementations of the FRAM using various formal tools have been published, for instance adding formal model checking [6]. They are however not yet able to deliver the level of automation we wish for, namely computing MSQ from the model. In addition the current simulator for FRAM models requires extensive manual effort to specify the variability behaviour of system functions as this information is not stored within the model but hand-provided at the beginning of each simulation.

The authors of [7] propose to use model-based safety assessment (MBSA) to perform the risk analysis of UAS concept of operations. They illustrate it on a loss of separation situation between a UAS and another aircraft. In order to model this scenario, they use the ADF language, and leverage its formal semantics to generate minimal cutsets leading to a mid-air collision. While the model itself is not explicitly stated to represent a procedure, the described workflow closely resembles one. We build our approach on this work, using its representation of failure propagation through the system by means of data-flow components.

B. Connecting models for safety

The approach of connecting models for increased safety insight is tackled by [8]. In order to maintain consistency between the MBSE system models and the MBSA safety models, they investigate the allocation of a system's functional architecture on its physical architecture. They describe several ways to specify that allocation, using either synchronisation, functional flows, physical resources or nested functions. Using the physical resources method, they found that it restricted the propagation of failures to the functional layer only, and that the resulting safety model unexpectedly grew very similar in structure to the system model. The question of using a mixed approach combining several of the presented allocation mechanisms remains open. We explore the use of multiple of the allocations mechanisms they describe in order to implement the connection between our two models.

The work presented in [9] introduces the notion of *Tiered Model-Based Safety Assessment*. It consists in formalising the relations between the safety assessments performed on three distinct views of the system model, called layers. These layers consist in a physical model, a functional model, and an operational model. Each layer thus corresponds to a given safety analysis while retaining the behaviour of the system. Our approach leverages this work and builds an additional model layer for the procedures. This extends the scope of the model to include not only the UAV but also other actors that might play a part in the execution of procedures.

III. Background

This section presents some background on procedures and introduces the AltaRica DataFlow (ADF) language we used.

A. Structure of a procedure

Operational procedures for UAVs are typically text-based documents. The procedure itself is a set of actions to be executed in a specific order by actors of the system. The authors of [7] model a scenario in which a procedure is carried out. They also add a set of detection tasks to the procedure execution tasks. Doing so allows them to account for the possibility that some actors might fail to recognize that execution tasks must be performed in the presence of a procedure-triggering event. This adds more failure scenarios and thus increases the potential benefit of computing the minimal set of failures, so-called *minimal cutsets*, leading to a given failure condition on their model.

In [1], we proposed a methodology for modeling the execution of a multiple procedures scenario. To this end, we extended the structure presented in [7] to account for the fact that each procedure has its own triggering events and to derive a consistent state for the system when several procedures are executed. This extended structure is as follows:

1) Initiation section : this section contains the events that require the application of a procedure. If one of these events is triggered, then the appropriate procedure must be executed.
2) Detection section : this section contains information about which actors can be made aware of the events of the initiation section, and whether they can fail to detect them, and in which way.
3) Execution section : this section describes the procedure tasks to be executed to achieve the stated goal of the procedure.
4) Outcome section : this section implements the desired rationale for computing the status of the system based on the previous sections across all individual procedures.

The procedures *stricto-sensu*, as described in the flight manual, correspond to the sole execution section. The other sections were added to make the model executable and analyses possible. In the remainder of this article, *the procedure* will refer to all four sections of the model.

B. AltaRica DataFlow

Systems are modeled in AltaRica DataFlow (ADF) as a set of interconnected components. ADF components contains the following elements, as illustrated in figure 1 :

- *flow variables* : these input and output variables represent the interface of the component;
- *state variables* : these internal variables represent state information, typically failure modes;
- *assertions* : they compute the output values based on the values inputs and state variables;
- *transitions* : these describe how the component's state variables can change;
- *events* : they trigger the *transitions* under the proper circumstances;

More precisely, transitions are said *fireable* when their associated guard (a condition over the values of the state and flow variables) is true. If a fireable transition's event is triggered, that transition is fired. Once fired, the values of the component's state variables are updated according to the actions defined by the transition. Finally, the assertions use the new values of the state variables to compute the new values of the output flows.

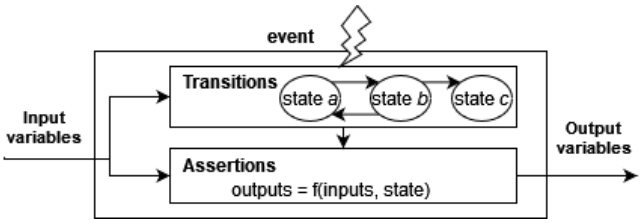

Fig. 1: Model of an AltaRica component.

IV. Use-case

The procedures we modeled and analysed are based on a small UAS. The source material for the procedures is the UAS's flight manual. Those procedures were modeled in the AltaRica DataFlow language, using Satodev's Cecilia Workshop software [10].

A. General presentation

Our use case is a fixed-wind UAV with a 3 meters wingspan, typically flying at 80 $km.h^{-1}$ to survey linear infrastructures.

The pilot may take manual control of the UAV using either the main communications channel (CC communications) or a back-up radio communications channel (RC communications). The availability of CC communications enables the pilot to engage one of several automated flight modes for the UAV to cruise on autonomously:

- *Flight plan* or *Resume* (*R*): the UAV flies its preset flight plan, possibly resuming it as it was before an emergency procedure was executed;
- *Go home* (*G*): the UAV returns to its home point and holds there;
- *Land* (*L*): the UAV lands automatically;
- *Manual control* (*MC*) : the pilot takes manual control of the UAV (usually performing a manual landing afterwards);
- *Flight termination* (*FT*): the UAV impacts the ground at low energy at end of a controlled downward-spiraling trajectory.

B. UAV system model

The UAV system model is based on the one presented in [9]. It consists in two layers : physical and functional, each of them representing a different view of the UAV. It was made using the ADF language, as were the procedure models we'll connect it to.

1) Functional layer: The functional layer can be split in two, as shown in Figure 2 :

First are the ADF components representing the acquisition functions for various flight parameters. These components have an input representing the execution status of the hardware they rely on, as allocated on the available CPUs. Additional inputs may be specified to account for other dependencies. More details about this can be found in [9].

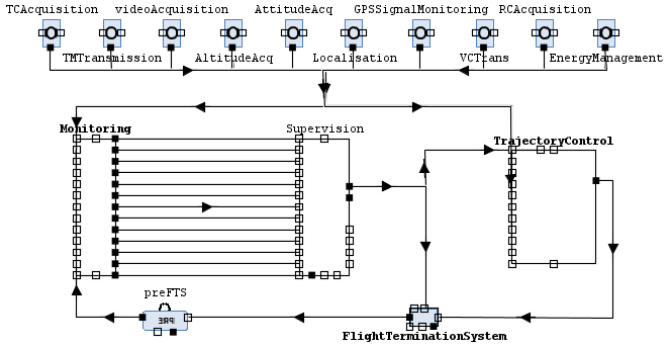

Fig. 2: Architectural model of the UAV (functional layer).

Second is the model of the on-board control-loop of the UAV. It is made of three main components. The *Monitoring* component watches the status of the acquisition functions' outputs. It produces the relevant alarm signal when it detects a failure among those. These signals then feed into the *Supervision* component.Based on what alarms have been raised, it determines the adequate flight mode for the UAV. The *Supervision*'s output feeds into the *Trajectory* component along with the outputs of the acquisition functions. These are used to determine whether the UAV is able to maintain itself within its flight envelope given its functional status and current flying mode. This information itself is in turn fed into the *Monitoring* component. An additional *FTS* component (Flight Termination System, in Figure 2) is activated by a specific output value of the *Supervision* component, and produces an output representing the activation status of the UAV's *FT* flight mode.

2) Physical layer: Shown in Figure 3, the physical layer provides the resources required to execute the UAV's functions. It contains ADF components representing the actual components of the UAV itself, both hardware and software. These represent sensors, actuators, CPUs, batteries and the

autopilot software of the UAV. It also displays the output of several components being allocated on either of the two on-board CPUs. The outputs of the CPUs represent the execution of the software task associated to the relevant ADF components on that CPU. These outputs are then carried over to the functional layer.

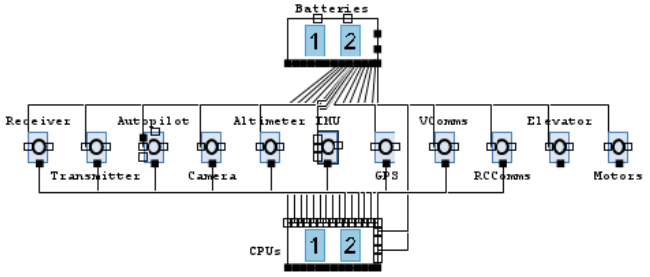

Fig. 3: Architectural model of the UAV (physical layer).

C. Procedure models

1) Sample procedures: We have selected three of the emergency procedures found in our UAV's flight manual. These procedures are named after the situation for which they are designed.

AP Fault (*APF*): a fault occurs within the auto-pilot; while the auto-pilot reboots, the UAV switches flight modes to *FT* in case that the auto-pilot does not recover. After a succesful recovery however, the UAV engages the *G* flight mode.

CC COMM LOSS (*CCL*): CC comms are disrupted, on either the uplink or downlink (RC comms are unaffected by this event); the UAV switches to *G* while the pilot monitors the UAV's trajectory using a specific video feed. The pilot takes manual control of the UAV after it arrives at its home point, and lands it himself.

F/CTL FAULT (*FCR*): the controller for the UAV's rudder fails; an emergency notification is sent from the UAV to the pilot while the UAV engages the *FT* mode. If aware of the emergency, the pilot takes manual control and must compensate for the UAV's lack of control to land the UAS as safely as possible.

2) Procedure model: Figure 4 shows one of the procedure models we later extend with the UAV model. That model was made using the methodology presented in [1]. It models the *CCL* procedure described above. The *CCL* component is the initiating event, which can be detected by either the pilot (*CCL_Detect_PIC*) or the UAV (*CCL_Detect_UAV*). The procedure then executes based on the detection status and the initial conditions. Components *CCL_G* and *CCL_Man* respectively represent the activation of the *Go home* or *Manual control* control modes.

V. Types of connections

There are two broad categories of mechanisms that can be used to connect elements of our models. The work in [8] describes similar mechanisms for connecting models through what they call allocations. However, they intend to use only one at a time for all model connections. Our approach is

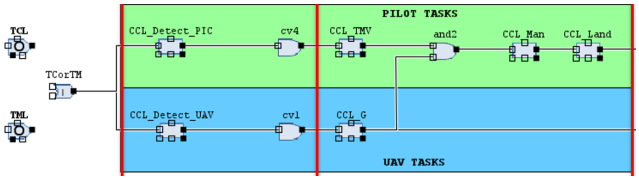

Fig. 4: ADF model of the CCL procedure.

to explore which mechanism is best adapted based on the element to connect, so we do not limit ourselves to a single type.

A. Dependency

this connection describes the functional dependency of an element *A* on another element *B*. *A* requires an input from *B* in order to produce an output value. This dependency can be further refined with regards to the behaviour of *A* on receiving an input from *B*. Formally, this distinction is not on the connection itself, which is specified by textually or graphically connecting two ports together, but rather on the behaviour of the component on the receiving end of that connection.

- *value dependency* : *A* uses the input from *B* directly for computing its output values;
- *state dependency* : *A* can be led to perform state transitions upon receiving specific values from *B*.

A *value dependency* in ADF is first specified graphically, as shown in figure 4. For instance *CCL_Man* depends on *CCL_Land* through its input. The specific behaviour associated to this dependency is then described within the dependent component itself, using the ADF code syntax as shown in the excerpt below:

```
current_exec = case I = Idle : Idle,
[...] else Success;
```

In this example, the output value of the component, $current\_exec$ is computed based on a pattern matching (keyword `case{}`) of the component's input and state variables. The code above shows that if the input I is received with value $Idle$, the component produces the output value $Idle$ as well. If I differs from $Idle$, the pattern matching continues with other values and variables.

We can also refine the *dependency* connection based on its necessity:

- *exclusive dependency* : *A* needs this input from *B* only;
- *optional dependency* : *A* needs this input from one of several input sources, of which *B* is but one.

In ADF, an input port can only be connected to a single output port. In order to get around this, we use a typical approach of aggregating several inputs through a modeling artefact component, as shown in figure 5. The component *TCorTM* produces a *Boolean* value representing whether or not either the TC or TM functions of the drone are non-nominal, and feeds this output into the procedure detection components monitoring this condition.

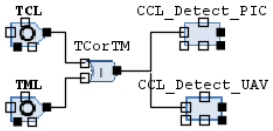

Fig. 5: ADF implementation of an optional dependency.

B. Synchronisation

A synchronisation connects two model elements through events. It allows to tie together the occurrence of several events across the model. There are several types of synchronisations, however we only present here the two types we used for our models.

- *strong synchronisation* : in a strong synchronisation H of events A and B, H is available if and only if both A and B are available. Firing H triggers both A and B. Neither A nor B can be fired independently of the other.
- *soft synchronisation* : firing a soft synchronisation S of events A and B triggers any event synchronised in S that is fireable. If A is fireable, but not B, firing S will fire A but not B. As for a strong synchronisation, none of the synchronised events are fireable independently of S.

VI. Connection Methodology

With these connections in mind, we can now look at our models and search for matching sections and elements between them. In doing so, we will be identifying which elements to connect using the previously described patterns, and which elements can be set apart as 'unrelated'. This methodology is aimed at connecting various types of models to a procedures model that follows a given structure We will thus use that structure to facilitate the identification of matching sections and components. Figure 6 shows an overview of the connections we made for the CCL procedure. They are described in more detail below.

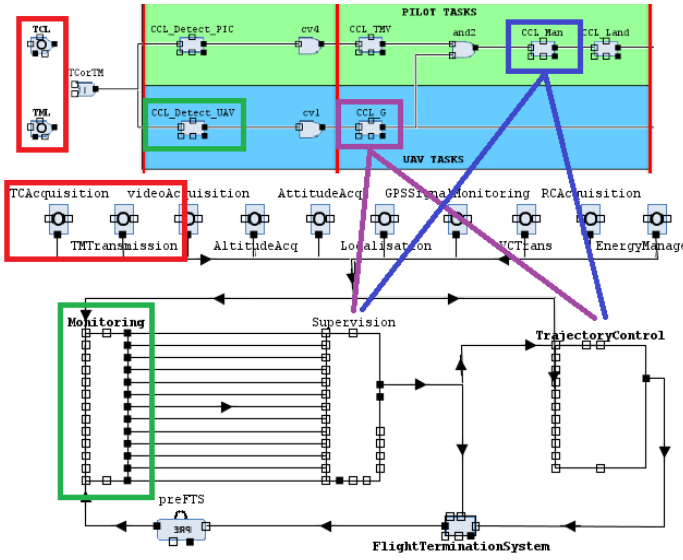

Fig. 6: Overview of connections between the CCL procedure model and the UAV system model.

A. Initiation section

any model element from the additional model whose failure corresponds to a procedure-initiating event can thus be matched to the initiation component modeling that event. The CC COMM LOSS (CCL) procedure described in IV-C1 is initiated by the loss of either the telecommand (TC) or telemetry (TM). Those functions are represented in our UAV model by the *TCAcquisition* and *TMTransmission* components respectively (highlighted in red in figure 6). Those functions are in turn allocated to the physical layer, respectively on the *Receiver* and *Transmitter* components (figure 3). The failure of any of these four components must thus trigger the CCL procedure.

Since we are considering failures events, it might seem relevant to use *synchronisations* to connect those components. In that case, we should then connect the CCL procedure initiating event to our four components' respective failure events. However, since the initial state of all components is the nominal state, the transitions labeled by the *failure* event is fireable. Indeed the guard of this transition is that the component's state is that the component be nominal. As such, regardless of the synchronisation we choose to implement, firing that synchronisation of our five events would fire all five events, meaning both TC and TM are lost simultaneously, across both the physical and functional layer. This introduces a coupling that is not realistic.

Thus, we must use a *dependency* connection (also called *driver-driven*. The *dependency* connection is directional : there is a component that sends an output flow, and a component that receives it as input. We will say that the receiving component is *driven* by the sending component, which *drives* the receiving component. In order to understand which component should be driven in a *dependency* connection, we have to remember why we are connecting models. The purpose of this approach is to model the impact of the system on the procedures. Failures occurring within the system were modeled at a high degree of abstraction in the procedures model, and were given a more concrete representation in the system model. As a consequence, we want the failures stemming from the more detailed model, the system model, to drive the failures occurring within the more abstract model, which is the procedures model. Thus, the driven component here is the CCL procedure initiating event, being driven by the four failure events of the system model.

Since there are four driving events here, we must use an *optional dependency* connection between each TM/TC system component and the CCL procedure initiating component. That way, the CCL procedure initiating event will activate when either of the four TM/TC system components fails. When those components fail, they produce a non-nominal output value. We thus implement a behaviour in our CCL initiating component, where if any of the four values it receives from the TM/TC system components is non-nominal, it activates.

If a procedure initiating event can be traced to the failure of a single other component, it might be thought that this

connection might then be implemented using a *synchronisation*. For instance, the APF procedure (loss of auto-pilot) could be linked to the failure of the only autopilot component in our model. However, the autopilot component in the UAV model might produce an erroneous output without failing, for instance if its resource itself fails. This would result in a situation where the APF procedure should be activated because the autopilot has been lost, but where it isn't because that loss does not result from a failure of the aupilot itself. This emphasizes the need of implement connections for failure events using the *dependency* connection.

B. Detection section

corresponding elements in the additional model represent monitoring, alarm or watchdog functions and/or components. In this case, we can choose either a synchronised or dependency approach.

Failure events can be synchronised between two matching alarms in each model, once established that the failure events lead to consistent output values in both models. This typically requires that failure modes and output values domain between the two alarms are similar. In addition, it also requires that any dependency one alarm might have on another component be also a dependency of the second alarm. For instance, consider the situation where a procedure alarm has no dependency (other than the component it monitors), and its corresponding system alarm has a dependency to another system component representing that alarm's power supply. If the power supply component fails, the system alarm produces a non-nominal value, but that will be not be matched by the procedure alarm. The later will still produce a nominal output as neither it or the component it monitors have failed. In order to ensure the validity of that synchronisation, the procedure alarm component must be made functionally dependent on the power supply component of the system model in the same manner as the system alarm is.

We can also use the *dependency* connection. The procedure alarm can be made directly dependent of the system alarm, by altering its behaviour so that it copies the output value of the system alarm to the corresponding value of the domain value for the procedure model. This simplifies the connection as it requires only one dependency compared to the previous approach which requires at least one synchronisation and as many dependencies as the system alarm has. However, the functional status of the driven alarm might end up being inconsistent with the driving alarm. Indeed, if the system alarm fails, this affects its output, which then affects the procedure alarm's own output through the flow connection. The procedure alarm's how internal state does not change and thus remains nominal, which makes it inconsistent with that of the system alarm.

In figure 6, the *Monitoring* component contains a set of alarms, each of which monitors one of the inputs of the component (these inputs are the outputs of the function components directly above). In order to connect the procedure alarm *CCL_Detect_UAV* to the corresponding alarm of *Monitoring*, we use ADF code to specify the adequate flow connection, and adapt the behaviour of *CCL_Detect_UAV* using code as described in V-A.

C. Execution section

Due to the high diversity of both the procedure tasks and additional model elements that can be modeled, the correspondence of procedures to additional model elements based on the execution section is harder to define. It must be performed with great attention to the nature of each execution task from the procedures model. In our case, the execution tasks describing the activation of a specific flight mode correspond to the execution status of the *Trajectory* component under the assumption that the *Supervision* component has commanded the flight mode corresponding to the original component in the procedures model. This illustrates how correspondences are not necessarily trivial and must reflect the system's behaviour. The privileged connection here is the *dependency* connection. We use it to implement two main connections.

First are the information flows, going mostly from the pilot (represented through the pilot tasks of the procedures model) to the UAV (the system model). Those typically represent the pilot giving the UAV a specific command, such as "Switch to manual control". In this type of *dependency* the pilot task is the driving component. When the *Supervision* component of the UAV receives a manual control command (represented by the successful execution of the corresponding pilot task in the procedures model), it produces an output to reconfigure the UAV in accordance with the pilot's order. As the current flight mode is stored in a state of the *Supervision* component of our system model, this *dependency* leads to a state transition and is thus a *state dependency*.

The other main connections in the execution section are the execution status of flight modes executed by the UAV. These are sent from the system model where the *Trajectory* component computes the quality of the trajectory based on the current flight modes and available UAV functions. They are received by procedure tasks. Those components are driven by the *Trajectory* component and reflect the UAV's performance of a given flight mode, they are *value dependencies*.

D. Outcome section

The outcome section of our procedures' structure is entirely a modeling artefact, and performs calculations based on the the execution status of the previous sections. As such it may have an equivalent in the additional model if the latter already contains modeling artefacts performing similar calculations. At this point, it is not relevant to connect those sections.It might however be relevant to compare their results in order to perform some consistency checks.

For instance, in our UAV's architectural model, the *Trajectory* component can determine whether the UAV remains in its flight envelope or if it deviates from it. If our procedures model indicates an all-clear outcome while the *Trajectory* component indicates an out-of-envelope trajectory, we know our model is inconsistent.

E. Unrelated components

Finally, not all components in either model necessarily corresponds to an element in the other model. Some elements in the additional model do not connect to any in the procedures model yet they are still required because they are essential to the additional model itself. Elements of the procedures model may not connect to the additional model, for instance because they would belong to another model. Typically, procedure task components representing tasks performed by the pilot may not have any corresponding components with our UAV's functional architecture, but would connect to a second additional model describing a human factors perspective of the pilot.

VII. Results and analysis

Our connected models contains 162 individual components, 62 for the procedures (of which 26 can fail) and 106 for the UAV (of which 42 can fail). 119 lines of code complete the specification of flow patterns. This enables us to represent three procedures connected to their UAV model. The Cecilia workshop can generate fault trees and compute relevant failure probabilities (provided that individual components are given quantitative failure rates). However, our primary interest rests in computing the qualitative safety indicator of minimal sequences (MSQ). These are the smallest sequences of events that can lead to an undesired event. We illustrate the reach of such an analysis by computing MSQ for the *Crash* outcome of our procedures-based scenarios, taking into account behavior of the UAV's functional architecture.

A. Model validation

Before we can perform this analysis, we must validate the connection of our models. This consists in verifying that our models behave consistently with regards to each other. To do so, we execute a scenario using the Cecilia workshop's built-in step-by-step simulator. We manually trigger an initiating event and ensure that model correspondences behave as expected. We also check for inconsistencies that may occur between the two models. Such an inconsistency manifests as our procedure and system model providing diverging results regarding the outcome of a failure. For instance, if at the end of a simulation the *Supervision* component commands the UAV in the *GoHome* flight mode and the *Trajectory* component indicates the the UAV can maintain its flight path, but the procedure model indicates that the UAV is crashed, there is an inconsistency in our models. Thus, we want to ensure that commands issued from the procedure are treated accordingly in the UAV model, and conversely, that detection and execution data sent from the UAV model to the procedures are processed as expected.

B. Computation and analysis of the Minimal Sequences

Table I below shows the sequences computed in 30 seconds on a 8-core CPU clocked at 2.5 GHz. We computed the sequences leading to the *Crash* outcome for each procedure individually, then for the three procedures together. We first computed sequences on the procedures model alone, *i.e.*, without taking account the contribution of the connected model. We computed the sequences taking into account the functional layer of our UAV model. Finally, we ran the computation on the procedures model and the physical layer of the UAV model.

No results are shown for sequences containing only one failure as there are none. Indeed, each procedure requires at least one event to be initiated. As such, for the outcome of any procedure to be *Crash*, at least one other failure as to occur, leading to the absence of 1st-order sequences.

Adding up the number of sequences leading to the crash of individual procedures may not yield the indicated subtotal, as is the case for the procedures and physical layer analysis. That is because a same sequence causes more than one procedure to end in the *Crash* outcome. Therefore, these sequences are identified as oMSQ for each individual procedure.

Reviewing the computed sequences provides several levels of information. First of all, the size (so-called *order*) of the shortest sequences as well as their number provides a first idea of the procedure's robustness, which can be used in order to assess its overall safety.

		Order of cut sequence	
Analysis scope	**Initiating fault**	**2**	**3**
Procedures alone	AP fault	2	8
	CC comm loss	8	4
	F/CTL fault (rudder)	2	24
	Total	12	34
Procedures and functional layer	AP fault	36	16
	CC comm loss	48	32
	F/CTL fault (rudder)	6	32
	Total	90	80
Procedures and physical layer	AP fault	58	78
	CC comm loss	86	220
	F/CTL fault (rudder)	23	56
	Total	145	336

TABLE I: Number of minimal sequences for each procedure.

Having the detailed sequences of event leading to undesirable outcomes may help guide designers' effort to improve the safety of their procedures and/or system. In the sequence {*TMTransmission.loss*, *Elevator.err*}, the loss of telemetry from the UAV (*TMTransmission.loss*) leads it to return above its home point. The failure of the telemetry later prevents the UAV from warning the pilot when the elevator gets stuck (*Elevator.err*), which then prevent the pilot from performing a successful landing, and crashes the UAV.

The number of sequences itself can also be leveraged to a useful purpose. By counting the number of sequences each specific events appears in, and by identifying to which component or subsystem each event belongs to, critical sections of the system may be identified, thus enabling efforts to be prioritized where most needed.

Our connected models could also be leveraged to perform an analysis similar to the *Failure Mode Effects and Analysis* (resp. *Failure Mode Effects Summary*) when focusing on the failures of the physical layer's components (resp. the impacts of the failures of the physical layer's components on the functional/procedure layers). We remind that an FMEA consists in reviewing exhaustively the failures of a system's

components and subsystems, and asses their impact on it. Using our connected modes, we can explore automatically the effect of simple failures of procedures, functions or physical items, whether nominal or emergency, and thus add a complement of information that might not be trivial.

C. Lessons learnt

1) Scope of the model: As things stand, our models thus provide an account of the UAV's functional and physical architecture impact on the safety of operational procedures. However, some inconsistencies persist. They are similar in nature to what is described in VII-A, and stem from the limited number of procedures we modeled. Furthermore, we have only considered the impact of the UAV's architecture, but as mentioned earlier, risk contributors in an UAV operation are many and diverse. Connecting additional safety models representing in detail the pilot or the support services used by the UAS would bring new failure modes and failure propagation paths.

2) Modeling process: In addition, we have introduced sequential behaviour in our model, in order to be able to compute the outcome of the scenario as well as to follow the propagation of flows step-by-step. This however also introduced the need to establish a priority relation between the transitions of our model components. This aspect of our methodology requires further work as a different priority model could induce significant changes in the dynamic of the model, and thus impact the results yielded by the safety analysis.

While using the ADF language to implement the methodology we presented on our use-case, we have found some limitations. ADF lacks some inheritance and genericity features that would have been particularly useful. Indeed, changing domain values or even simply adding a value to a domain requires significant rewriting effort across most of the components that use that domain. This slows down both the modeling and analysis process. Using higher order programming languages, model elements could be specified much more generically, both in terms of structure and behavior. The ALPACAS project [11] proposes a DSL based on Scala and could be the foundation of such an effort.

3) Analysis: Finally, we focused our modeling efforts in providing the support necessary to perform safety analyses based on the computation of MSQ, which are a quantitative safety artefact. While those are indeed a relevant safety indicator in many regulations and standards, qualitative safety are of paramount importance. The Cecilia Workshop we used provides some support for quantitative safety analysis, but we did not pursue this objective. Thus, further work is required on that front to yield satisfactory quantitative results from our current methodology.

VIII. Conclusion

A UAS is composed of heterogeneous elements whose failures during the operation are multiple and diverse in nature. Understanding how these failures contribute to the risk of an operation is essential to assess the safety of a UAS, and may help UAS designers to comply with regulator's requirements.

Having identified operational procedures as a keystone of any UAS operation, we proposed a methodology to connect domain-specific safety models relating to the UAV operation to the procedure models we developed. Doing so allows us to bring more detailed knowledge of a specific section of the system and assess how it contributes to the risk of the operation. We implemented this methodology on a medium-sized UAS using the AltaRica DataFlow language. Using the Cecilia Workshop built-in tool, we illustrated the benefits of our approach by explaining how automatic analyses like minimal sequences computation can support safety analyses.

However, our methodology does not yet preclude entirely the possibility of inconsistencies between two connected models. Further work is thus required in order to identify and eliminate those, as well as to extends the scope of the analyses we perform to include quantitative safety indicators. Furthermore, the implementation of our methodology in the AltaRica DataFlow language emphasised the benefit to be had from using higher-order functions and mechanisms such as genericity and inheritance.

Acknowledgment

This work was supported by the Defense Innovation Agency (AID) of the French Ministry of Defense (research project CONCORDE N° 2019 65 0090004707501).

References

[1] C. Mathou, K. Delmas, J.-C. Chaudemar, and P. de Saqui-Sannes, "Modeling uas flight procedures for sora safety objectives," in *2023 IEEE International Systems Conference (SysCon)*. IEEE, 2023, pp. 1–8.

[2] S-18 Aircraft and Sys Dev and Safety Assessment Committee, *Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment*, dec 1996. [Online]. Available: https://doi.org/10.4271/ARP4761

[3] T. Ishimatsu, N. G. Leveson, J. Thomas, M. Katahira, Y. Miyamoto, and H. Nakao, "Modeling and hazard analysis using stpa," 2010.

[4] E. Hollnagel and O. Goteman, "The functional resonance accident model," *Proceedings of Cognitive System Engineering in Process Plant*, 01 2004.

[5] Q. Yang and J. Tian, "Model-based safety assessment using fram for complex systems," in *Saf. Reliab. Complex Eng. Syst.-Proc. 25th Eur. Saf. Reliab. Conf. ESREL*, 2015, pp. 3967–3974.

[6] Q. Yang, J. Tian, and T. Zhao, "Safety is an emergent property: Illustrating functional resonance in air traffic management with formal verification," *Safety science*, vol. 93, pp. 162–177, 2017.

[7] P. Bieber, C. Seguin, V. Louis, and F. Many, "Model based safety assessment of concept of operations for drones," in *Congrès Lambda Mu 20 de Maîtrise des Risques et de Sûreté de Fonctionnement*, St Malo, France, 10 2016.

[8] M. Machin, E. Saez, P. Virelizier, and X. de Bossoreille, "Modeling functional allocation in altarica to support mbse/mbsa consistency," in *Model-Based Safety and Assessment: 6th International Symposium, IMBSA 2019, Thessaloniki, Greece, October 16–18, 2019, Proceedings 6*. Springer, 2019, pp. 3–17.

[9] K. Delmas, C. Seguin, and P. Bieber, "Tiered model-based safety assessment," in *Model-Based Safety and Assessment: 6th International Symposium, IMBSA 2019, Thessaloniki, Greece, October 16–18, 2019, Proceedings 6*. Springer, 2019, pp. 141–156.

[10] "Cecilia workshop." [Online]. Available: https://satodev.com/en/our-products/cecilia-workshop/

[11] M. Buyse, R. Delmas, and Y. Hamadi, "Alpacas: a language for parametric assessment of critical architecture safety," in *35th European Conference on Object-Oriented Programming (ECOOP 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.