Flight Incident Analysis Through Symbolic Argumentation

Dionisio de Niz[1], Bjorn Andersson[1], Mark H. Klein[1], John Lehoczky[1], Hyoseung Kim[2], George Romanski[3], Jonathan Preston[4], Floyd Fazi[4], Daniel Shapiro[5], Douglas C. Schmidt[6], Ronald Koontz[7], and Sam Procter[1]
[1]Carnegie Mellon University, dionisio@sei.cmu.edu, baanderson@sei.cmu.edu, mk@sei.cmu.edu, jl16@andrew.cmu.edu, sprocter@sei.cmu.edu;
[2]University of California Riverside, hyoseung@ucr.edu;
[3]Federal Aviation Administration, george.romanski@faa.gov;
[4]Lockheed Martin Corporation, jonathan.d.preston@lmco.com, floyd.a.fazi@lmco.com;
[5]Institute of Defense Analysis and University of California Santa Cruz, daniel.g.shapiro@gmail.com;
[6]Vanderbilt University, d.schmidt@vanderbilt.edu;
[7]The Boeing Company, ron.j.koontz@boeing.com.

***Abstract*— At the core of every modern airliner is a software-reliant fly-by-wire system that translates pilot inputs into electronic signals to control aircraft movements. Given the safety-critical nature of these systems they include architectural constructs and mechanisms to tolerate failures related to hardware (e.g., processor or sensor failures) and software (e.g., potential bug in the code). The goal is to reach the required levels of availability and integrity validated through a certification process that includes specific verification methods to discharge specific claims. Unfortunately, the different verification procedures and associated architectural constructs are typically developed independently and make independent assumptions that can contradict each other, thereby preventing the desired behavior or invalidating the assumptions and results of a given verification procedure. To help address these problems this paper presents how a new symbolic argumentation approach can be used to analyze a real flight incident (the flight CI202 incident in 2020) by automating the verification procedures and their assumptions. Our approach describes verification plans that start at the level of certification connected to automated verification analysis on architectural models. These plans are decomposed into analysis contracts that specify what claims they verify (e.g., availability of a fly-by-wire function > 99.99%), what analysis is used to verify the model (e.g., probabilistic Fault-Tree Analysis) and what assumptions it relies on (e.g., a function is replicated over processors that fail independently of each other). These plans are integrated into a symbolic argumentation implemented as a constraint satisfaction problem that is solved with a Satisfiability Modulo Theory (SMT) solver. The CI202 flight incident analysis is presented using an argumentation hierarchy on architectural models and the analysis of potential design issues that could explain a triple computer failure. We demonstrate how our approach can reason about early design decisions by pointing to unfulfilled assumptions, contradictions, and potential workarounds that have the potential to prevent these types of incidents.**

Keywords—component, formatting, style, styling, insert *(key words)*

I. Introduction

Safety-critical software-reliant systems like avionics systems must satisfy strict safety properties that are verified through certification. To meet these properties (e.g. availability), different architectural constructs (e.g., replication) and mechanisms (e.g., fail-over) are used in conjunction with verification procedures (e.g., probabilistic Fault-Tree Analysis). We call this triple combination an *assurance architectural construct (AAC)*. Unfortunately, different assurance architectural constructs are develop independently making equally independent assumptions. As a result, when multiple AACs are used together within a system keeping track of the assumptions and their potential interactions become a risky manual activity.

This paper discusses the flight incident of the flight CI202 that occurred in Taiwan in 2020 where a triple computer failure left the pilots braking in semi-manual mode. We present how we used a new symbolic argumentation approach that allows us to model AACs and how the certification claims are decomposed into arguments composed of contracts that describe what different verification procedures (in the form of analysis) verify, what they assume and how we verify such assumptions and their interactions. With this approach we can model the problem described in the incident report [16], describe the assumptions that caused the problem and automatically analyze them, and explore potential solutions.

A. Related Work

This paper shares some objectives with [1]. In particular, they promote the concept of assurance-based development that is similar to the concepts we present here. However, their focus is on the final system; hence, they focus on the application of verification tools to the final system. In contrast, we focus on a more comprehensive approach that encompasses early designs and higher levels of abstractions where arguments can be developed and verified much earlier. These higher-level abstractions are closer to the higher-level requirements that document intended behavior at a higher-level granularity. Moreover, the low-level focus closer to the code forces abstractions that remove aspects from other analysis domains (e.g., timing, fault tolerance) thereby creating blind spots for conflicts in analysis assumptions from these domains.

The techniques we use are presented in [2], which is based on analysis contracts that describe what an analysis tries to prove, what assumptions it makes, and how it connects to a full

argument developed to prove verification claims. Analysis contracts rely on assume/guarantee reasoning based on Hoare triples [3], which evolved into more abstract domains with the development of contract algebras [4]. Contracts have also been used in assume/guarantee reasoning over components in the Architecture Analysis and Design Language (AADL) [5][6]. However, component contracts reason about properties of the values that AADL components communicate through their ports to other components and the computation that occurs inside a component that transforms input values into output values. This approach is a more traditional way of thinking about contracts that is easier to map to a Hoare triple. In contrast, analysis contracts reason about properties of analysis algorithms applied to models (e.g., AADL models), not the computations inside the components of the model. Our goal is to reason about how multiple analyses work together to prove top-level assurance claims instead of how properties on values generated by model components discharge properties of top-level components. From this point of view, therefore, an analysis that uses component contracts to verify value transformation properties is just another analysis that we integrate and that would have its own analysis contract.

Previous work in analyses contract started with [7], where contracts were defined for resource allocation models. These contracts were defined in Alloy [8], and the analyses algorithms were implemented in Mathematica and included in the AADL models. Analyses contracts were later extended [9] to remove the bounded verification limitations of Alloy, implementing the contracts specification with a mixture of satisfiability modulo theories (SMT) and linear-time temporal logic (LTL) [10] with a verification in Z3 and SPIN [11]. This work also extended the analyses beyond resource allocation to other domains, such as thermal dissipation and security. Later, the authors in [12] created an implementation of analysis contracts with a special emphasis on lower-level analysis assumptions within the same domain.

In [13], the authors present a contract model close to analysis contracts with a synthesis approach to combine multiple contracts that restrict the design space out of pre-crafted parts. Their approach works at a more abstract level closer to [4]. It is applied at the assurance case level and reuse of assurance case patterns but provides no connection to domain-specific analysis algorithms.

Given the focus on using tools for analysis, it is natural to ask whether to trust that the output of the analysis was computed correctly. The recently-coined term explainable verification focuses on addressing this issue [14][15]. This effort focuses on the approach that an analysis not only needs to compute an output but also an explanation of why it produced this output. This explanation must then be easy to consume by a person without requiring deep expertise in the analysis domain.

II. A Flight Incident Case

The flight incident we analyze in this paper exemplifies the challenges we face when multiple certification claims, such as availability, integrity, and timeliness, need to be addressed by multiple assurance architectural constructs. This incident occurred in the Taipei airport in 2020. The core of the incident is described in [16] as follows:

> *On June 14, 2020, China Airlines scheduled passenger flight CI202, an Airbus A330-302 aircraft, registration B-18302, took off from Shanghai Pudong International Airport for Taipei Songshan Airport with 2 flight crew members, 9 cabin crew members, and 87 passengers, for a total 98 persons onboard. The aircraft landed on runway 10 of Songshan Airport at 17:46 Taipei local time. At touchdown, the aircraft experienced the quasi-simultaneous failure of the 3 flight control primary computers (FCPC or PRIM), thus ground spoilers, thrust reversers, and autobrake were lost. The flight crew was aware of the autobrake and reversers failure to activate, and applied full manual brake rapidly to safely stop the aircraft about 30 feet before the end of runway 10 without any damage to the aircraft nor injuries to the passengers onboard.*

The incident report identifies the lack of synchrony in redundant computations within the system as the main culprit. To understand the situation we will use Figure 1. which depicts a partial view of the flight control architecture of the Airbus 330.

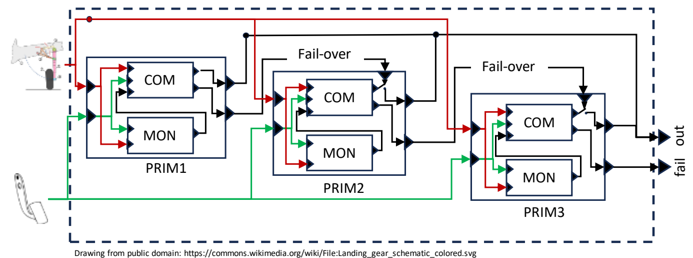

Figure 1 Airbus A330 Flight Control Architecture

Figure 1 shows the decomposition of each PRIM computers into a main command processor (COM) (part of a sequence of processors or channels as known in aviation) and a monitor processor (MON). Both the COM and MON processors calculate the same actuation value according to the pilot input (depicted by a pedal in the figure). After both compute this value, COM receives the computed value from MON and compares it to its own value. If the difference is within a threshold, it uses its output. If the difference exceeds the threshold, however, it fails-over to the next PRIM computer (PRIM2) arranged in the same fashion. The PRIM2 computer performs the same comparison and has the same failover strategy. If PRIM3 also fails, the backup computers are activated.

It is important to note that these computers execute different functions when the aircraft is in air mode versus when it is in ground mode. This is because in air mode the vertical tail is actuated whereas in ground mode, the front wheels are actuated.

A. Key Sequence of Events

In the incident report, the following sequence of events was identified as significant:

1. Gear touchdown → Aircraft switch to ground mode
2. Gear in the Air →Aircraft switch to air mode

Identify applicable funding agency here. If none, delete this text box.

3. Gear touchdown →Aircraft switch to ground mode
4. PRIM1 Failed-over
5. PRIM2 Failed-over
6. PRIM3 Failed-over

The report also identified that the cause of the failure stemmed from the disagreement between the COM and the MON over the actuation in [16] (p. 68). This disagreement was traced back to the computation of different functions (controls laws) by COM and MON due to the switching be-tween air mode and ground mode. In particular, one of the channels used the "lateral flight control law" to calculate the rudder command (based on the pilot pedal input), while the other used the "lateral ground law," exceeding the threshold that was designed to compare differences for the same law [16] (p. 6). This disagreement was called "channel asynchronism."

B. Channel Asynchronism Timeline

Based on the information presented above we created a sample timeline that could explain the triple failure, which is shown in Figure 2.

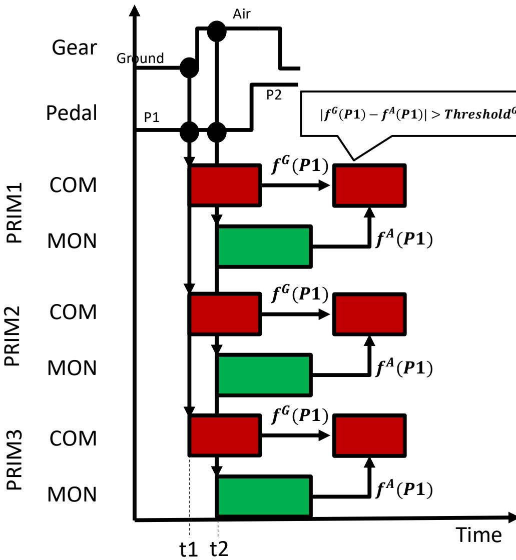

Figure 2 - Channel Asynchronism Timeline

Figure 2 shows one specific time sequence that, according to the report, can occur in the system and would explain the triple failure. Specifically, the figure shows all COM processors (from the three PRIM computers) reading the landing gear (and pedal) position at the same time at time $t1$. This occurrence is then followed by the reading of the gear and pedal position by all MON processors at time $t2$. Given that the plane is in ground mode at time $t1$, all COM processors use the "lateral ground law" (identified as $f^G(P1)$ in the figure) to calculate the rudder command. On the other hand, all MON processors then use the "lateral flight law" (identified as $F^A(P1)$ in the figure) to calculate the rudder command. The MON command is then communicated back to the COM processor, and it calculates the difference and compares it to the threshold ($|f^G(P1) - f^A(P1)| > Threshold^G$)

Clearly, while MON and COM use the same pedal input ($P1$) they use different functions to calculate the output, violating the implicit assumption of the comparison (i.e., they compare results of the computation from the same control law). Interestingly, the two computations were performed in the same time frame. The difference exceeded the threshold and PRIM1 fails-over to PRIM2. However, since PRIM2 and PRIM3 suffer from exactly the same flaw, they both fail as well. The end result is that all three MON processors from the three PRIM computers evaluated that they exceeded the tolerance threshold, and all three executed a failover that led to the switching to the secondary computers.

Note that Figure 2 shows only one possible sequence of events that could lead to this incident. In reality, any sequence that leads COM and MON to execute different control laws will lead to the same incident.

III. AUTOMATIC CERTIFICATION ARGUMENTATION

At the core of this paper is the need to verify if we can use automatic argumentation methods that automate the verification procedures attached to certification arguments. This section presents the techniques from our current work that we use to implement this automation and identify possible design flaws connected to the CI202 incident report.

A. Symbolic Assurance Refinement Framework

To enable the automatic argumentation, we used the Symbolic Assurance Refinement (SAR) framework and tool [2]. This framework allowed us to integrate analyses defined for different properties, such as timing, fault tolerance, control, security, etc. into an argumentation structure that validates their assumptions and their interconnections with other analyses. More specifically, an analysis can make assumptions (e.g., tasks are scheduled with fixed-priority scheduling with rate-monotonic priorities) that must be true for the result of an analysis (e.g., rate-monotonic schedulability bound) guarantee (all threads always meet their deadlines) to be valid. Checking these assumptions can involve more complex analysis that would also need to have its assumptions checked for the analysis' guarantee to hold. This argumentation is depicted in Figure 3.

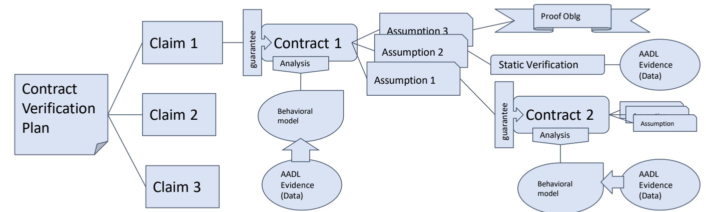

Figure 3 - Contract Argumentation

SAR specifies the analysis contracts in a domain-specific language hosted as a language annex in the AADL tool (OSATE[1]). This template is presented in Listing 1.

```
annex contract {**
    contract <name> {
    queries
        <model var> = <query to obtain model data>
    domains
        <domain reference>
    input assumptions
        <Bool func to check data consistency>(<model
vars>)
    assumptions
        <Bool func>(<model vars>)
        -> <symbolic assertion>
    analysis
        <Bool func>(<model vars>)
        -> <symbolic guarantee>
    }
**};
```

Listing 1:Analysis Contract Template

Analysis contracts have three main parts: (1) a guarantee that is encoded symbolically (in SMT in the current implementation), (2) assumptions that are assertions also specified symbolically, and (3) an analysis that takes the form of a Boolean proposition that can be implemented as an imperative function that takes model data (model variables) and verifies specific conditions that would make the guarantee true. For instance, checking if a taskset is schedulable using the rate-monotonic harmonic taskset bound implies only checking if the sum of the utilization of all the tasks running in a single processor is below 100 percent (i.e., a simple test like $\sum_{i \in taskset} \frac{WCET_i}{Period_i} \leq 1$). However, here the main challenge is checking all the assumptions of the analysis, specifically

1. All tasks are periodic. In the model (e.g., AADL model) this can be a flag but would need to be checked in the implementation.
2. Periods of the tasks are harmonic (i.e., are multiples of each other). $\forall\, i, j \in \text{taskset} : (\text{Period}_i > \text{Period}_j) \rightarrow \text{Period}_i \bmod \text{Period}_j = 0$
3. Priorities are rate monotonic. $\forall\, i, j \in taskset : (Period_i < Period_j) \rightarrow Priority_i > Priority_j$ (A larger priority number implies a higher priority.)
4. Periods are equal to their deadlines.
5. Tasks are scheduled with a fixed-priority scheduler.
6. Tasks do not use mutually exclusive resources. This can be a more complex search in the model for shared resources and the protocols used to access them. This can be an independent contract whose implementation can have further assumptions.

Contracts have three additional sections that complement the main ones:

- queries (similar to SQL queries) that collect data from the architectural model for use by the analysis (Queries are, in fact, expressed in a domain-specific model query language [MQL].)
- domains that are basically the names of a separate contract module that defines symbolic variables to be used in the assumptions and guarantee statements (Some of these variables can mirror model variables, but others will be used to encode a property even if they never appear in the model. For instance, the rate-monotonic harmonic bound test guarantees that the worst-case response time of no task will exceed its deadline under any circumstance. For this, we define a worst-case response time even if it is not in the model and is never calculated in the scheduling test but is in the proofs of the paper that demonstrated the correctness of this analysis.)
- input assumptions that validate the data obtained in the queries to check if there is enough data to run the analysis (These are different from the other assumptions [we call analysis assumptions] in that if we do not have enough data [i.e., input], then we cannot run the analysis function. However, if the analysis assumptions are not met, then the result would be invalid even if we have enough data.)

B. *Proof Obligations and Refinement*

SAR enables two forms of analysis assumption verification. First, SAR verifies the existence of model data that will make the assumption false. For instance, if the model already has priorities and periods assigned to tasks, then we can check if the assumption can be violated with this data. If we do not have data that contradicts the assumption, then we can assume that it is correct. This assumption is implemented by the SMT engine that determines if there are symbolic variable values that can satisfy all our assumptions. This determination is useful for the verification of partial models when we do not have everything specified yet. However, assumptions verified this way are considered **proof obligations**, which are basically deferred obligations to verify assumptions.

Second, SAR verifies that the data we have or do not have would not enable an assignment that would contradict the assumption. For instance, if the model has no information about priorities, we will verify if we can find values for the symbolic variable that would make some assumption false. This verification assumes that we now must have all the information to validate all the claims and assumptions. Hence, if this verification fails, it means that we still have proof obligations to fulfill, and we need to refine the model to verify the pending assumptions/claims. This type of verification is formally known as ensuring that a formula is valid (for all assignments).

The first type of verification allows us to verify partial models. This, in turn, allows us to keep refining the model and verifying each refinement step. Finally, once we believe we have

[1] OSATE stands for *Open source AADL Tool Environment*.

a complete model, we can then use the second type of verification to validate whether we are truly done.

C. Assurance Argumentation

The contract argumentation starts at the top from a verification plan to capture verification claims and the analysis contracts that can discharge them. These contracts follow the argumentation structure presented in Figure 3. The verification plan structure is presented in Listing 2 along with an example contract for end-to-end timing analysis.

```
annex contract {**
  verification plan verifyEndtoEndTiming {
    component
      s: EndToEndTimingExample::mysystem.i;
    domains
      schedulability;
    claims
      `And([E2EResponses[i] <= E2ELatencies[i]
        for i in range(len(E2EResponses))` ;
    contracts
      EndToEndDelayedCommunicationContract;
  }

  contract EndToEndDelayedCommunicationContract {
    domains
      schedulability;
    queries
      input assumptions
        '''areEndToEndLatenciesInputDataComplete(
              ${periods$},
              ${wcets$},
              ${deadlines$},
              ${names$})''';
    assumptions
      contract areConnectionsDelayedContract;
      argument schedulabilityArgument;
    guarantee
      <=> `And([E2EResponses[i] <= E2ELatencies[i]
        for i in range(len(E2EResponses))])`;
    analysis
    '''meetEndToEndLatencies(${flowComponents$},
                          error0)''';
  }

  argument schedulabilityArgument {
    domains
      schedulability;
    guarantee
      <=> `And([Deadlines[i] >= Responses[i]
            for i in range(len(Deadlines))])`;
    argument
      or(
        contract RMAHarmonicBoundContract
        contract RMANonHarmonicBoundContract
        contract fpResponseTimeContract
      );
  }
**};
```

Listing 2:Verification Plan

Listing 2 shows the verifyEndToEndTiming verification plan that includes the contracts (only the EndToEndDelayedCommunicationContract in this case) that are used to verify all aspect of the claims presented in the verification plan. (This is an SMT encoding of the claims.)

In the description of the EndToEndDelayedCommunicationContract in Listing 2, we see that its assumptions are verified with another contract (areConnectionsDelayedContract) and an argument (schedulabilityArgument). The schedulabilityArgument argument enables the selection of different forms of verification of assumptions (different scheduling analysis) that enables the selection of different forms of verification of this assumption (different scheduling analysis).

The details of the `schedulabilityArgument` are also presented in the listing and show how it is possible to combine multiple contracts into a Boolean formula, or, in this case, to use any of the contracts that are possible to use. In this particular case, it will use the `RMAHarmonicBoundContract` if it can verify its assumptions, which includes both that the priority assignment is rate monotonic and that the periods of the task are harmonic to each other. If it cannot verify the period harmonicity, the **OR** encoding then allows us to try to use the `RMANonHarmonicBoundContract` contract that does not require period harmonicity. Finally, if neither of the two assumptions are met, it is then possible to use the `fpResponseTimeContract` contract that does not require either of the two but requires other assumptions (e.g., the deadline is shorter or equal to the period).

IV. Assurance Argumentation for the A330

While we lack the specific claims of the original certification of the A330 aircraft and the role of the replication patterns (with the PRIM computers and their COM/MON processors), we first explore a generic form of the replication to identify potential claims, verification procedures, and assumptions to develop an assurance argumentation.

A. Replication Patterns

The A330 architecture has two forms of replication that, to our understanding, address different properties. In particular, one type of replication addresses reducing the possibility of calculating the wrong value or preserving the value integrity (or just Integrity for short). The other goes after preserving the availability of the computation even if faults occur; this is known as Availability.

1) Integrity Replication

From the incident report, we believe the A330 is implementing integrity replication with the combination of the COMMAND module (COM) and the MONITOR module (MON) that calculate the same output from the input commands, and the COM uses the MON data to check its own computation.

Assumptions

The integrity replication has at least two assumptions:

1. Development diversity requires that two different teams develop two (or more) modules independently. The rationale behind this approach is that if we develop two different implementations that can fail differently, we would be able to detect the failure of the pair based on the difference in the output values. This diversity assumption can be checked with the proper execution of a development process that enforces it, but the incident report does not point to a failure related to this assumption.
2. Module pair should use the same input, which means that both COM and MON should receive the same input to create the proper comparison between the two. More importantly, in the specific case covered by this paper, while it is possible to get some small variation in the input from the pedal sensor, a difference in the air/ground mode

is catastrophic given that it selects different control laws for the same module. This behavior is exactly what the incident report describes as a key violation.

2) Availability Replication

Availability replication is used to tolerate hardware failures and preserve a module to continue running. Availability claims aim to satisfy measures of tolerance to failures in the form of either some informal argument on failure independence or a more formal probability of the absence of service due to the failure of all the replicas.

Assumptions

Key to the availability replication is the assumption that the modules must fail independently. More specifically, this assumption means that the hardware where the modules run must fail independently. For this paper, we focus only on this assumption given that we believe it informs the design of the avionics system and interacts with the integrity replication scheme.

B. *Replication Modeling*

We first formalize the availability claim (probability of absence of service) connected to a specific verification procedure: probabilistic fault-tree analysis (FTA). In this case, we create the replicated end-to-end flow architecture (that captures the channel construct) that reads from the sensors, computes the actuation in the PRIM (COM/MON) computers, and sends it to the actuator as shown in Figure 4. This figure depicts with each box an independent thread (including the sensors and actuator boxes), representing the last thread that interacts with the appropriate device (i.e., reads from the sensor registers or writes to the actuator registers) that runs in its own processor. For simplicity, we assume that all processors fail independently.[2]

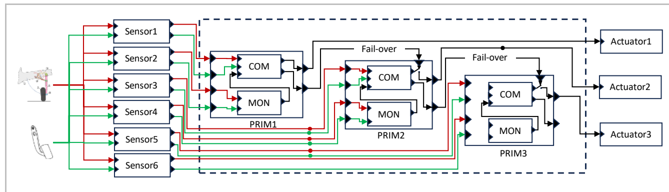

Figure 4: All Independent Channel Threads

1) Fault-Tree Analysis (FTA)

FTA is a top-down approach that defines the faults of interest under which a tree of events is created that lead to such a fault [17]. The tree is constructed by first connecting abstract events to the top-level failure by either an AND or an OR connector to represent that for the fault to occur, either all the next level abstract events need to occur or only one of such events needs to occur respectively. The decomposition of the next level events continues in the same fashion until basic events are identified. This construct can also be translated into what is known as a reliability block diagram, where OR compositions are represented as blocks connected in series while AND are those connected in parallel. These patterns are shown in Figure 5.

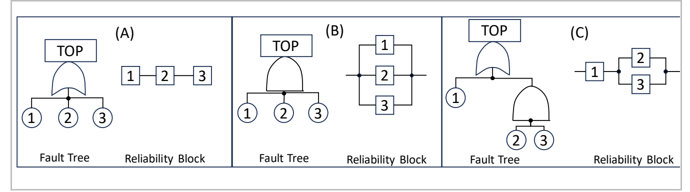

Figure 5: Fault-Tree and Reliability Block Diagrams

Fault trees allow two types of analysis: a qualitative analysis and a quantitative analysis. The qualitative analysis allows us to evaluate the set of basic events that, if they occur simultaneously, makes the top fault occur. These are known as the cut set. A cut set is minimal if it cannot be reduced without losing its status as a cut set.

In addition, a fault tree also allows us to identify common causes that break a failure-independence expectation. For instance, the minimal cut sets of the systems in Figure 5 are {1},{2},{3} for system A, {1,2,3} for system B and {1}, {2,3} for system C. These sets allow us, for instance, to identify single points of failures as cut sets with a single element. In our example, both system A and C have them, but system B does not.

The quantitative analysis, on the other hand, allows us to calculate the probability of failure by assigning probabilities of occurrence to each basic event and deriving the probability of occurrence of the top event from them. This probability is calculated with

$$Q_o(t) = 1 - \prod_{1 \le j \le k} (1 - \check{Q}_j(t))$$

Equation 1

with $\check{Q}_j(t)$ as the failure probability of cut set C_j calculated as

$$\check{Q}_j = \prod_{i \in C_j} q_i(t)$$

Equation 2

where $q_i(t)$ can be calculated as a single event occurring at time t (after some time of service) that is not repairable or is repairable and needs to be calculated over time. For the objective of this analysis, we assume that it is not repairable and that $q_i(t)$ is given; we apply it in the A330 architecture to evaluate potential claims and their assumptions.

2) AADL Model

[2] Further verification is required for the final implementation e.g., power supplies, cooling systems, etc.

To capture the replication characteristics to perform the FTA, we created an AADL model [5] following the architecture in Figure 4[3] where each component is a thread. We next created a hardware architecture where each thread has its own processor assigned to it. We then identify the internal control/data flows (identified as f<number>) that cross each component from input to output ports, the connections from output to input ports (identified as c<number>), and the flow source (identified as s<number>) and flow sinks (identified as k<number>). This architecture is shown in Figure 6.

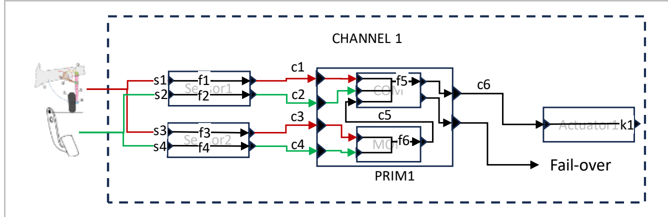

Figure 6: Channel Composition

We now discuss two more details added to the AADL model that we could not include in the figure:

1. All ports in this model are *Data* ports, which means that when the thread activates, it reads whatever is in the port buffer and continues its execution.
2. All connections are *Delayed* connections, which means that whatever the thread in the output port of the connection sends, it arrives at the next periodic activation of the receiving thread.

This combination allows us to (1) abstract away the network communication delays and assume that the communication will happen within the execution of the previous periodic activation and (2) assume that when the receiving thread activates, it already has the most recent data in its input ports.[4] Given all this information, we can define two end-to-end flows, as presented in Listing 3.

```
pedalToActuationCOMPRIM1: end to end flow
Sensor1.s1-> Sensor1.f1->c2->COM.f5->c6->
Actuator1.k1;
pedalToActuationMONPRIM1: end to end flow
Sensor2.s3-> Sensor2.f3->c4->MON.f6->c5->
COM.f5->c6->Actuator1.k1;
```

Listing 3: End-to-End Flows Replica 1

Listing 3 only lists one flow in each component because all the flows are equivalent given that their data is available at the time of the thread activation. Moreover, the end-to-end flow pedalToActuationMONPRIM1 captures the dependency between COM and MON that is not present in the end-to-end flow pedalToActuationCOMPRIM1. These characteristics of Listing 3 allow us to focus only on the MON end-to-end flows when describing the replication pattern.

To describe the replication pattern, we define that the main flows within each of the PRIM computers must be replicas of each other. In Listing 4, we capture only the pedalToActuationMONPRIMx given that it captures the internal dependencies between the MON and COM processors.

```
ReplicationProperties::Replicating =>(reference
(pedalToActuationMONPRIM2), reference
(pedalToActuationMONPRIM3)) applies to
pedalToActuatorMONPRIM1;
```

Listing 4: Replication Specification

In addition, we specify the probability of failure of each processor and the target reliability (or availability) of the replicated flow as presented in Listing 5. (Note that this probability is notional and does not represent the typical requirement of a commercial aircraft.)

```
ReplicationProperties::ReliabilityTarget => 0.85
applies to pedalToActuationMONPRIM1;
ReplicationProperties::FailureProbability => 0.01
applies to Sensor1.processor;
```

Listing 5: Reliability Specs

We developed a probabilistic FTA analysis that uses Equation 1 and Equation 2. In this analysis, we transform the graph created with the end-to-end flows into a reliability block diagram based on the processor each thread runs in and their dependencies. Two observations are in order. First, COM depends on MON given that it uses its output to evaluate output value differences; therefore, there is no advantage to running them in separate processors since if one stops working (e.g., is not available), the other will not be able to verify its output and will fail as well. Second, for the COM comparison with MON, it needs to read the pedal and landing gear at exactly the same time since it is not possible to know exactly when the mode switch will happen. This new requirement is captured in Listing 6, which shows only the PRIM1 part.

```
ReplicationProperties::
  IntegrityReplicas =>(reference
    pedalToActuationCOMPRIM1) applies to
    pedalToActuationMONPRIM1;
ReplicationProperties::
  ReplicasStartJitterTolerance => 0 ms applies to
     pedalToActuationCOMPRIM1;

Period 100 ms applies to Sensor1.thread;
Period 100 ms applies to Sensor2.thread;
```

Listing 6: Simultaneous Activation

Listing 6 captures a new type of replica that we call IntegrityReplica; it identifies the replicas used to evaluate if two functions implemented to compute the same value really do so. As discussed above, to work properly, these replicas require the same input values (within a tight tolerance). We also added the

[3] Not shown for brevity.

[4] Clearly, both assumptions need and can be verified with a more detailed model, but this discussion is not included in this paper.

periods for the sensor threads so that we can calculate the worst-case jitter of their respective threads (i.e., the worst-case possible difference between the starting of the execution and, hence, sensor reading) of the thread that starts the flow. (We left this equal to zero.)

We were able to satisfy the Reliability target with our FTA analysis, but we also discovered that the COM and MON threads do not benefit from running on independent processors given that if either COM or MON fails to produce the correct value, both fail together. For the jitter analysis, however, we realized that we cannot have two different sensing threads for COM and MON if our jitter target is zero. Hence, we modified the architecture as presented in Figure 7 with both MON and COM using a single thread to read the sensor values at the same time.

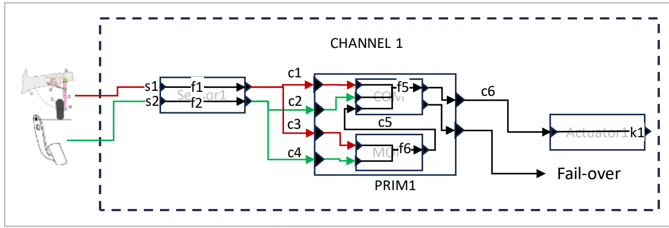

Figure 7: Channel 1 With Single Sensing Thread

Finally, we evaluated the end-to-end latency of the flows to verify that they meet the proper requirement by adding all the schedulability parameters for each of the threads, specifically

- periods
- deadlines
- priorities
- worst-case execution time
- assignment of threads to processors (where they will run and already used for the reliability replication)
- scheduling policy

In Listing 7, we introduce the properties for only one thread given that the assignment to the other threads follows the same pattern.

```
Period 100 ms applies to Sensor1.thread;
Deadline 100 ms applies to Sensor1.thread;
Priority 1 applies to Sensor1.thread;
Compute_Execution_Time 10 ms applies to
Sensor1.thread;
Actual_Processor_Binding =>(reference
(rep1.com.proc))
   applies to Sensor1.thread;
Preemptive_Scheduler => True applies to
rep1.com.proc;
```

Listing 7: Scheduling Properties

C. *Automatic Certification Argumentation*

By applying the Symbolic Assurance Refinement (SAR) framework described in Section III.A, we developed the verification plan for the CI202 incident presented in Listing 8. This listing starts with the verification plan specification module at the top. It then presents three contracts used in the plan to verify the claims and assumptions for integrity (SamplingSynchronizationContract), availability (ReliabilityContract), and end-to-end timing (EndToEndDelayedCommunicationContract). In the claims section of the verification plan, we also have three SMT claims about the three claims that we verify.

```
annex contract {**
  verification plan verifySynchronization {
    component
      s: EndToEndTimingExample::mysystem.i;
    domains
      synchronization;
      reliability;
    claims
      `And([E2ESamplingJitter[i] <=
         E2ESamplingJitterTolerance[i]
         for i in
            range(len(E2ESamplingJitter))])`;
       `And([Reliability[i]>=ReliabilityTarget[i]
          for i in range(len(Reliability))])`;
        `And([E2EResponses[i] <= E2ELatencies[i]
          for i in range(len(E2EResponses))` ;
     contracts
       SamplingSynchronizationContract;
       EndToEndDelayedCommunicationContract;
       ReliabilityContract;
  }
  contract SamplingSynchronizationContract {
    domains
      synchronization;
    guarantee
      <=> `And([E2ESamplingJitter[i] <=
            E2ESamplingJitterTolerance[i]
            for i in
range(len(E2ESamplingJitter))])`;
    analysis

   '''areFlowsInSync1(${flowComponents$},
     error0)''';
  }
  contract ReliabilityContract {
    domains
      reliability ;
    assumptions
    '''areReplicasOnIndependentProcessors(
        ${flowComponents$},
        error0)''';
    guarantee
      ⇔ `And([Reliability[i] >=
          ReliabilityTarget[i]
          for i in range(len(Reliability))` ;
    analysis
      '''isE2EFlowProbFail Met(
            ${replicatede2es$},error0)` ;
  }
  contract EndToEndDelayedCommunicationContract {
    domains
      schedulability;
    queries
    input assumptions
      '''areEndToEndLatenciesInputDataComplete(
       ${periods$}, ${wcets$},
       ${deadlines$}, ${names$})''';
    assumptions
      contract areConnectionsDelayedContract;
      '''areAllThreadsPeriodic(${threads$},
        ${protocols$},
        ${names$},error0)'''=>
        `And([Periodics[i]
          for i in range(len(Periodics))])`;
      '''areAllDeadlinesConstrained(${threads$},
```

```
            ${periods$},${deadlines$},
            ${names$},error0)''' =>
            `And([Deadlines[i] <= Periods[i]
            for i in range(len(Deadlines))])`;
      argument schedulabilityArgument;
    guarantee
      <=> `And([E2EResponses[i]
                <= E2ELatencies[i]
          for i in range(len(E2EResponses))])`;
    analysis
      '''meetEndToEndLatencies(
         ${flowComponents$},
         error0)''';
  }

  argument schedulabilityArgument {
    domains
      schedulability;
    guarantee
      <=> `And([Deadlines[i] >= Responses[i]
             for i in range(len(Deadlines))])`;
    argument
      or(
        contract RMAHarmonicBoundContract
        contract RMANonHarmonicBoundContract
        contract fpResponseTimeContract
      );
  }
**};
```

Listing 8: Verification Plan

The SamplingSynchronizationContract contract details are also included in Listing 8. The guarantee is expressed in SMT and is checked with the Python function areFlowsInSync1() in the analysis section to verify the ReplicaStartJitterTolerance presented in Listing 6.

Similar to the SamplingSynchronizationContract, the ReliabilityContract presented in Listing 8 includes the verification of the independence assumption that is verified with the Python function areReplicasOnIndependentProcessors(). This function returns True if it is able to evaluate that the functions in the end-to-end flows that model the different PRIM computers run on independent processors and False otherwise. (This end-to-end flows data is obtained from queries in MQL that obtain the data from the AADL model; they are not shown for brevity.) Then, in section analysis, an invocation to the Python function isE2EFlowProbabilityOfFailureMet verifies whether or not the reliability requirement presented in Listing 5 is met.

The last contract presented in Listing 8 is the EndToEndDelayedCommunicationContract. The details presented in the listing follow a similar pattern to the other two contracts with the following exceptions. First, it adds an input assumptions section with a Python call to evaluate whether we have enough data to run this contract. Second, it adds the contract areConnectionsDelayedContract as one of the assumptions to verify, which is one of the complex assumptions that uses another contract to verify it. Finally, it adds the argument schedulabilityArgument that enables the selection of different forms of verification of this assumption (different scheduling analysis), as discussed in the Assurance Argumentation in Section III.C.

V. Limitations

Our experiment was limited to validate the capabilities of the current techniques to investigate design errors in systems already deployed. To take full advantage of these techniques it is necessary to validate them in early design to prevent these errors from occurring. Similarly, it is also important to validate these techniques in systems modifications as the systems evolve. As more analyses techniques of different level of formalization are incorporated in the proposed argumentation we expect the automatic argumentation to evolve. More specifically, the symbolic constraints specified in the SAR approach are expressed in SMT first order logic, this means that this must be evaluated to a Boolean. However, on the one hand, engineering analyses at times use engineering judgement to evaluate whether some non-exhaustive analysis or probabilistic analysis yields a result that is acceptable. On the other hand the use of simulators, e.g., like in digital twins have proven to be useful even if they do not explore all possible behaviors of the system and no absolute Boolean value can be derived from it. In this exercise we translated probabilistic measurements (e.g., probability of failure) into a Boolean by ensuring that such probability reaches a threshold. Perhaps in the future a deeper integration of probabilistic analysis and non-exhaustive analysis can be explored.

VI. Concluding Remarks

The objective of this paper was to validate if the automation of certification arguments can indeed identify problems that occur in real systems. Key to this validation exercise was the following:

1. The connection of the arguments to certifiers and designer rationale is typical of certification reasoning.
2. The arguments can be expressed at a high enough level of abstraction to enable humans to evaluate whether the arguments are properly captured or not.
3. The verification of the arguments and the verification procedures of the claims in such arguments can be automatically processed by a computer.
4. The lower-level assumptions of the verification procedures (i.e., analysis) can be captured and verified at lower levels of detail, incrementally increasing the details getting all the way down to implementation but in a way that each level can be reasoned incrementally.

This work allowed us to validate all these aspects. However, limits to the incremental validation did not enable us to produce a full implementation given the lack of information in our example. At the same time, we understand that more work is required to validate if this approach can be extended to the scale of full certification claims and incremental recertification. Subsequent experiments will explore this issue in more detail.

We learned the following lessons from conducting the research presented in this paper:

- From a methodological point of view, this work allowed us to demonstrate how multiple abstractions used by different verification domains (real-time scheduling,

availability, and integrity) can be assembled and cross validated in an automatic, formal way. Importantly, these abstractions are connected to architectural models that are reasonably easy to understand by engineers and certifiers. These abstractions can also be connected to the results from the specific analysis (even if all of the details are not fully understood).

- This work allowed us to exercise the reuse of arguments (e.g., for end-to-end timing verification) that encapsulated multiple checks at lower levels of abstractions to which the modeler reusing the argument was not exposed. At the same time, new analyses were added (availability and integrity) along with their assumptions and assumption checking analyses; this allowed us to automatically identify failures to meet these assumptions in the fictitious design variations that we explored.

In summary, this modeling and automatic verification experiment showed a robust process that allowed us to encode lessons learned in an executable format. This encoding can then be reused as we develop new arguments and verification procedures that can be used in new system developments. More importantly, the encoding enables the automatic integration of development activities to the production of certification evidence. These types of automatic certification techniques are crucial to software-reliant systems, such as defense and aerospace systems, that continuously evolve [18]. To highlight this importance and encourage interest from industry and research communities, we have formed the Assurance Evidence for Continuously-Evolving Real-Time Systems workgroup (ASERTW). Please visit https://www.asertw.org/ to learn about our activities and follow-up research.

ACKNOWLEDGMENT

The following markings MUST be included in work product when attached to this form and when it is published.
For purposes of double anonymous peer review, markings may be temporarily omitted to ensure anonymity of the author(s).
Copyright 2024 Carnegie Mellon University, Daniel Shapiro, Douglas C. Schmidt, Floyd Fazi, George Romanski, Hyoseung Kim, John Lehoczky, Jonathan Preston and Ron Koontz
This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.
References herein to any specific entity, product, process, or service by trade name, trade mark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by Carnegie Mellon University or its Software Engineering Institute nor of Carnegie Mellon University - Software Engineering Institute by any such named or represented entity.
NO WARRANTY. THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.
[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution.
This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License. Requests for permission for non-licensed uses should be directed to the Software Engineering Institute at permission@sei.cmu.edu.
Carnegie Mellon® is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.
DM24-0780

REFERENCES

[1] Shankar, Natarajan; et al.. *DesCert: Design for Certification*. March 2022. https://arxiv.org/abs/2203.15178

[2] de Niz, Dionisio & Wrage, Lutz. Symbolic Refinement for CPS. ACM SIGAda Ada Letters. Volume 43. Issue 1. Pages 88–93. 2023. https://doi.org/10.1145/3631483.3631498.

[3] Hoare, C. A. R. An Axiomatic Basis for Computer Programming. Communications of the ACM. Vol-ume 12. Number 10. Pages 576–580. October 1969. https://dl.acm.org/doi/10.1145/363235.363259.

[4] Benveniste, Albert, et al.. Contracts for System Design. Foundations and Trends in Electronic Design Automa-tion. Volume 12. Issue 2-3. 2018. https://www.nowpublishers.com/article/Details/EDA-053.

[5] Architecture Analysis and Design Language (AADL). SAE International. Standard AS5506. March 2009. https://www.sae.org/standards/content/as5506/.

[6] Cofer, Darren; Gacek, Andrew; Miller, Steven P.; Whalen, Michael W.; LaValley, Brian; & Sha, Lui. Compositional Verification of Architectural Models. In NASA Formal Methods. Pages 126–140. 2012.

[7] Nam, Min-Young; de Niz, Dionisio; Wrage, Lutz; & Sha, Lui. Resource Allocation Contracts for Open Analytic Runtime Models. In 2011 Proceedings of the Ninth ACM International Conference on Embedded Software (EMSOFT). Pages 13–22. October 2011. https://doi.org/10.1145/2038642.2038647.

[8] Jackson, Daniel. Alloy: A Language and Tool for Exploring Software Designs. Communications of the ACM. Volume 62. Issue 9. Pages 66–76. August 2019. https://doi.org/10.1145/3338843.

[9] Ruchkin, Ivan; de Niz, Dionisio; Garlan, David, & Chaki, Sagar. Contract-Based Integration of Cyber-Physical Analyses. In EMSOFT '14: Proceedings of the 14th International Conference on Embedded Software. Pages 1-10. October 2014. https://doi.org/10.1145/2656045.2656052.

[10] Kesten, Yonit; Pnueli, Amir; & Raviv, Li-on. Algorithmic Verification of Linear Temporal Logic Specifications. In Automata, Languages and Programming. Larsen, K. G.; Skyum, S.; & Winskel, G. [editors]. Springer Berlin Heidelberg. 1998.

[11] Holzmann, Gerard J. The Model Checker SPIN. IEEE Transactions on Software Engineering. Volume 23. Number 5. Pages 279–295. 1997. https://doi.org/10.1109/32.588521.

[12] Brau, G.; Hugues, J.; & Navet, N. Towards the Systematic Analysis of Non-Functional Properties in Model-Based Engineering for Real-Time Embedded Systems. Science of Computer Programming. Volume 156. Issue 1. 2018. http://dx.doi.org/10.1016/j.scico.2017.12.007.

[13] Wang, Timothy E.; Daw, Zamira; Nuzzo, Pierluigi; & Pinto, Alessandro. Hierarchical Contract-Based Synthesis for Assurance Cases. In NASA Formal Methods: 14th International Symposium, NFM 2022. May 2022. http://dx.doi.org/10.1007/978-3-031-06773-0_9.

[14] 1st International Workshop on Explainability of Real-Time Systems and their Analysis (ERSA). IEEE Real-Time Systems Symposium (RTSS 2022). Houston, Texas. December 2022. https://sites.google.com/view/ersa22.

[15] 2nd International Workshop on Explainability of Real-Time Systems and their Analysis (ERSA). IEEE Real-Time Systems Symposium (RTSS 2023). Taipei, Taiwan. December 2023. https://sites.google.com/view/ersa23.

[16] Taiwan Transportation Safety Board (TTSB). China Airlines Flight CI202 Occurrence. TTSB-AOR-21-09-001. September 2021. https://www.ttsb.gov.tw/media/4936/ci-202-final-report_english.pdf.

[17] Rausand, Marvin. Reliability of Safety‐Critical Systems: Theory and Applications. John Wiley & Sons, Inc. 2014. https://onlinelibrary.wiley.com/doi/book/10.1002/9781118776353.

[18] de Niz, Dionisio, et al. Assurance Evidence of Continuous Evolving Real-Time Systems. Technical Report, Software Engineering Institute | Vanderbilt University | Federal Aviation Administration, 2022. https://www.andrew.cmu.edu/user/dionisio/pubdocs/ASERT-Report-Final.pdf.

[19] Bjorn Andersson et al., "Explainable Verification: Survey, Situations, and New Ideas," Technical Note, Software Engineering Institute at Carnegie Mellon University,https://insights.sei.cmu.edu/documents/5866/survey-explain_CGrLAVz.pdf