Assurance for Integrating Advanced Algorithms in Autonomous Safety-Critical Systems

Milton Stafford, Siddhartha Bhattacharyya, *Member, IEEE*, Matthew Clark, Natasha Neogi, and Thomas C. Eskridge

***Abstract*—Although advanced algorithms are needed to enable increasingly autonomous civil aviation applications, there are limitations in assurance technologies, which must be addressed to gain trust in the performance of these algorithms. This gap emphasizes the need to guarantee safety by capturing performance boundaries, as these algorithms are integrated. Additionally, multiple similar algorithms might need to be executed sequentially or concurrently to accomplish a mission or provide guidance for safety-critical operations. The selection among algorithm functionalities is a complex and critical activity that needs to be systematically designed and analyzed before actual implementation. Toward this end, we discuss our proposed process, which includes formally modeling abstractions of the algorithms in an architectural framework, then identifying the key performance parameters, followed by verification of the composition of these algorithms with formal contracts based on assumptions and guarantees. Finally, to reduce the gap between design and implementation, an automated translation from the architectural model to source code has been developed, which is a Java-based outline of the implementation. We demonstrate our compositional approach in assuring the behavior of an autonomous aerial system via a collision avoidance case study with advanced algorithms to handle critical emerging situations.**

***Index Terms*—Architecture design analysis, automated reasoning, autonomous system design, formal methods.**

I. Introduction

With the advent of machine learning and adaptive control (AC), there have been significant advancements in the field of intelligent systems. Many disciplines have seen the widespread adoption of advanced techniques, such as artificial intelligence (AI) and AC. These advanced technologies have enabled execution of highly complex tasks previously done by humans in diverse areas of application or solving previously unsolvable problems. The integration of advanced technologies has led to the increasingly autonomous systems (ASs) of today. These systems are detailed in several research efforts investigating AC [1]–[3] and AI-based methods [4]–[7]. For example, the automatic supervisory AC method enables the AS to fly with a damaged wing [2]. The automatic ground collision avoidance system (AutoGCAS) for fighter aircraft automatically prevents ground collision [8]. Path planners, such as Stratway [9] and Advanced ReRouter [10], deploy approaches to prevent airborne collision along with identifying a path from the origin to destination. The aforementioned algorithms are only a sampling and emphasize the need to integrate these complex methods in an assured fashion, so they can be deployed in commercial and military applications. The need for assurance means and methods is further emphasized in [11], which deals with certification considerations for adaptive systems.

The integration of advanced technologies within an AS enables the AS to intelligently avoid hazardous situations or to deal with emerging situations autonomously. Therefore, it is highly beneficial to deploy these capabilities for safety- and mission-critical operations. However, these technologies are not yet implemented for autonomous operations in safety-critical systems, due to challenges in assuring the correctness of behavior. Upon further inspection, a primary shortcoming in the high-assurance software environment is the lack of techniques to demonstrate consistency of complex automated systems. Consequently, there is a significant demand for integrating advanced AI/AC systems into safety-critical domains. In order to integrate complex algorithms, a formal structured approach needs to be established starting from the design phase so as to better understand the performance boundaries.

Considering the scenario in which multiple advanced software controllers are executing concurrently, the problem compounds. The complex nature of the problem demands a rigorous approach toward the design of a decision-making component (DMC). A high level of confidence on the AS is required, as the success, failure, and safety of the system depend on the DMC being able to select one or more of the relevant services to deal with the hazards of the emerging situation. In avionics, advanced software controllers could include existing modules such as AutoGCAS or hypothetical automated variants of a traffic alert and collision avoidance system (TCAS) and a geo-fence avoidance system. The problems stem from the possibility of multiple concurrent events or failures, requiring intervention from multiple controllers. This leads to the challenge of abstraction in representing

Manuscript received April 23, 2020; revised August 10, 2020; accepted September 5, 2020. Date of publication October 5, 2020; date of current version December 9, 2021. This work was supported by the Harris Institute of Assured Information, Florida Institute of Technology. *(Corresponding author: Siddhartha Bhattacharyya.)*

Milton Stafford was with the Department of Computer Science, Florida Institute of Technology, Melbourne, FL 32901 USA. He is now with Maxar Technologies, Westminster, CO 80234 USA (e-mail: mstafford2012@my.fit.edu).

Siddhartha Bhattacharyya and Thomas C. Eskridge are with the Department of Computer Science, Florida Institute of Technology, Melbourne, FL 32901 USA (e-mail: sbhattacharyya@fit.edu; teskridge@fit.edu).

Matthew Clark is with Galois Inc., Dayton, OH 45402 USA (e-mail: mattclark@galois.com).

Natasha Neogi is with NASA Langley Research Center, Hampton, VA 23666 USA (e-mail: natasha.a.neogi@nasa.gov).

Digital Object Identifier 10.1109/JSYST.2020.3023286

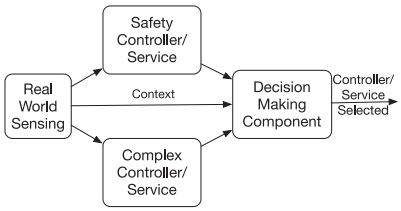


Fig. 1. Simplex architecture.

the complex algorithms and the DMC, so that we can formally verify each component and the composition. Finally, once the design is verified, the next problem to address is reducing the assurance gap between the design and its implementation. To address this concern, we developed an automated approach to translate the architecture and the formally analyzable annotations within the architecture to an outline of the Java-based software code. This becomes the outline for software engineers to follow and embeds the algorithmic behavior in Java. This article discusses a systematic approach to integrate advanced algorithms, through architecture analysis, by incorporating existing and novel methods for automated analysis. The process is elaborated by implementing the methods within a tool. The end result is the description of a process to integrate advanced algorithms.

The challenges in designing the models describing the abstract representation of the complex algorithms are set out in Section III. We discuss the design of the DMC and assess its responsibilities in Section IV. Finally, an algorithmic approach translating the architecture into software code is detailed in Section V. We then describe an autonomous aerial system case study, with an Architecture Analysis and Design Language (AADL)-based architectural representation and an assume–guarantee-based modeling of the abstracted behavior, in Section VI.

II. Background

This section on related work discusses simplex architectures. We focused on the simplex architecture as it provides us with the essential components that are needed in order to integrate the complex or untrusted algorithms required for AS to handle emerging situations. In the simplex architecture, shown in Fig. 1, one of the essential components is the DMC. Therefore, one of the major focuses of this article is in designing the DMC by architecturally analyzing and understanding the behavior of the complex algorithms. The analysis of the complex algorithms provides us with the specifications that need to be satisfied by the DMC. Additionally, we investigate if the actions from the complex algorithms can be selected sequentially or concurrently. Toward this end, we discuss the AADL, which is the architectural framework that we have used in designing the models. This is followed by a discussion of the Assume–Guarantee REasoning Environment (AGREE) that enables compositional analysis of the designed solution.

A. Simplex Architecture for Complex Algorithm Selection

Complex or uncertified controllers/services can be integrated within ASs for decision making by using a simplex architecture approach [12]. According to a simplex architecture [12], [13], the real-world situation observed with sensors is fed into the DMC, complex controller(s), and safety controller(s). The controllers compute their solution, and the DMC selects one based on the understanding of which controller performs the best in that context. Therefore, the design of the DMC is very critical as it selects the controller to take the next possible action.

The simplex mindset introduces the idea of concurrency, that is, a "high-assurance kernel" executing in tandem with the "high-performance subsystem" [14]. The emphasis of this article is in analyzing the performance of complex algorithms, at a higher level of abstraction, to understand the performance boundaries. This is guided by the motivation to prevent ASs failures such as the Tesla accident, where the performance boundary of the vision system was not well understood [15].

In [12], Phan *et al.* combined a simplex architecture with assume–guarantee-based reasoning to assure runtime behavior. However, Phan *et al.* [12] did not investigate handling multiple emerging situations, which is performed in this article. Wang *et al.* discuss the RSimplex [16] architecture, which integrates robust fault-tolerant control techniques into the simplex architecture, but it does not determine the performance boundaries to identify when the guarantees fail for a system.

B. Architecture Modeling With AADL

In the modeling environment, system developers create high-level models of the system architecture using the System Modeling Language [17], which is standardized by the Object Management Group. Another similar modeling paradigm is the AADL [18], which is supported by the SAE AS2-C working group.

AADL is a component-oriented modeling framework. A model is a set of interconnected component instances. A component instance is a unique representation of a component type that defines its interface (e.g., parameters, ports, requirement to access to other elements, etc.) and the corresponding component implementation that refines its internals. This implementation allows for a diverse set of system characteristics to be modeled as AADL annexes depending upon the automated analysis of interest such as scheduling, model checking, behavior as finite-state machines, and compositional reasoning. AADL has been defined so that a system/software engineer can use the vocabulary and concepts of his/her domain. AADL allows for integrating and reasoning over the composition of the architecture.

C. Modeling Behavior With AADL Annex: AGREE

The AGREE annex is an extension to AADL that allows the behavior of a component to be specified as assume–guarantee contracts. In AGREE, the assumptions specify the assumed inputs with its ranges and the corresponding outputs with its guarantees. AGREE analysis follows system engineering principles such as architectural reasoning to perform compositional verification. A component-level guarantee is met by guarantees provided by the subcomponents of that component. Tool support for AADL and AGREE is provided by the Open Source AADL Tool Environment (OSATE) [19]. The OSATE provides

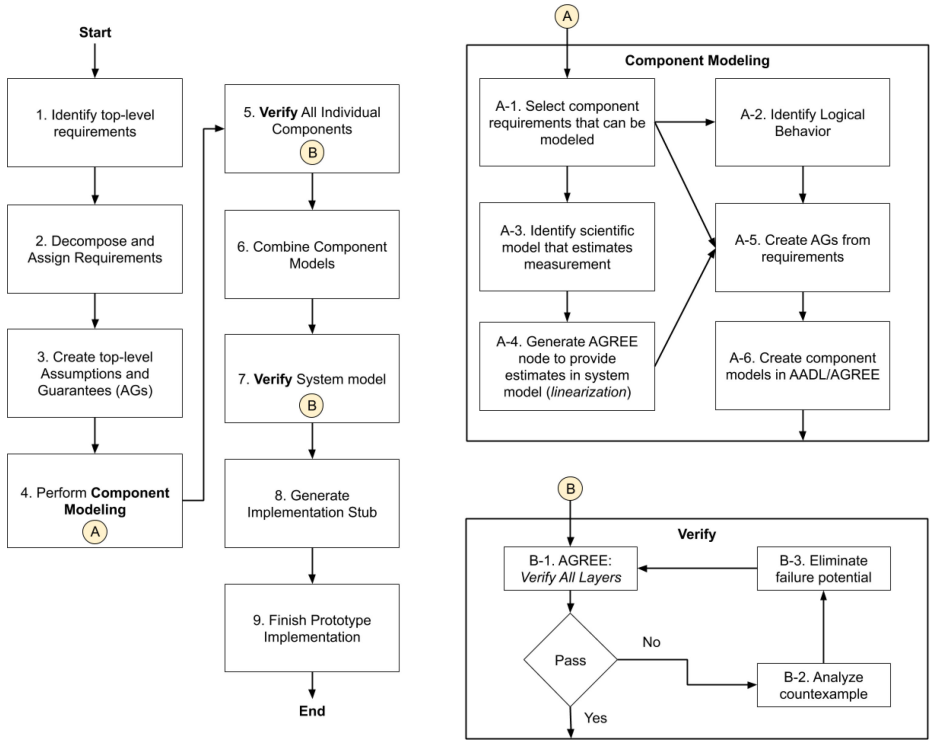


Fig. 2. Proposed framework methodology. The steps on the left form the main routine. Both enclosed blocks on the right are subroutines referenced by the main routine.

graphical and textual editors for AADL and supports a number of analysis tools.

In [20], researchers modeled the behavior of avionics as a behavior annex that was then translated into Uppaal [21] to perform model checking. Cofer *et al.* [22] developed models in AADL/AGREE to provide verifiable security, that is, system designs that provide the highest levels of confidence in their security based upon verifiable evidence. *Our major contribution is in the design of the framework to integrate advanced algorithms.* In the integration process, we demonstrate how an architectural assume–guarantee-based approach can be used to represent complex algorithms and to identify satisfaction of requirements in the design phase. As a result, we developed an algorithmic approach to generate key performance parameters (KPPs), implemented a method to identify the existence of concurrent solutions to emerging problems, and then generated the execution schedule based on the emerging situation. In implementing this framework, our article also elaborated on the design and verification of a DMC for integrating complex algorithms. Finally, we developed an AADL to Java translator that demonstrated how we can automate the process of mapping the design to source code generation with constraints modeled in the architecture.

III. Methodology for Integrating Advanced Algorithms

One of the essential elements in safely integrating advanced algorithms for ASs is to develop a systematic approach that emphasizes automated architectural analysis early in the design phase. The early automated analysis objective helps capture concrete requirements and perform automated checks to detect conflicts in the requirements or identify the uncertainties. Our methodology elaborates upon such an architectural approach.

Fig. 2 illustrates the entire methodology. The steps are given as one main routine (on the left) and two subroutines (on the right) referenced by the main routine. The remainder of this section discusses all the key elements of the proposed methodology, which involves identifying preconditions, decomposition, component modeling, verification, composition, and implementation and then elaborates on those steps.

A. Preconditions

The preconditions that need to be satisfied in order to apply our method requires knowledge of certain predefined objectives, predefined failures, and a model checking environment. Thus, there must exist an initial notion of desired behavior such that top-level requirements can be identified (Step 1 in Fig. 2). For example, a top-level system requirement for a TCAS may be for the rightmost of the two aircraft in conflict to ascend, while the other aircraft descends, in order to resolve the conflict. In other words, the highest level module must have a well-defined objective that also serves as a nonsubjective measure for failure. Along with the objective, one needs to identify an architectural modeling environment that allows automated reasoning to identify conflicts among requirements or guarantee performance.

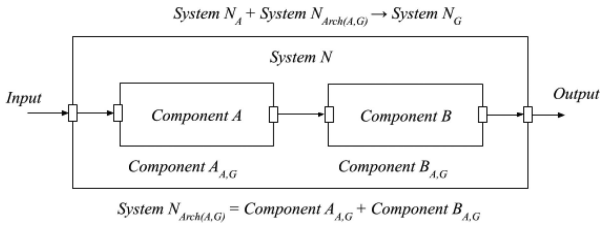


Fig. 3. Compositional reasoning.

B. Decomposition

The top-level requirements identified so far must be broken down into lower level requirements and assigned to components (Step 2 in Fig. 2). In other words, a sensible separation of responsibility is undertaken. The decomposition is an example of the time-honored tradition of breaking problems into subproblems.

In the process of decomposition, it is critical to identify the top-level or system-level requirements as assumptions and guarantees for the whole system (see Fig. 3). This top-level goal is paramount to the correct operation, so all submodules/subsystems should adhere to the assumptions of the system, which should lead to the satisfaction of the system-level guarantees. These system-level requirements are modeled in the AGREE language. For example, a system-level guarantee may be on the overall time taken by the system to execute a sequence of complex algorithms to address an emerging situation; this is further discussed in Section VI.

C. Component Modeling

Step 4 in Fig. 2 indicates that the *component modeling* subroutine is executed on each individual component, that is, the execution of steps A-1–A-6. The first step in modeling a particular software module is identifying local objectives and requirements. The three aspects of a component are the logical behavior (A-2), the scientific model estimation of the continuous dynamics (A-3), and the objective requirements. Identifying these aspects help us write the assumptions and guarantees in AADL/AGREE (A-5).

The logical behavior of a software module is the set of rules that model discrete behavior for actions within a module. For example, consider a software module that is responsible for an aircraft's altitude that picks 7000 ft if heading Eastward and 8000 ft otherwise. To model this behavior (A-2), you must introduce an input parameter, e.g., "`heading`," and model the altitude decision based on that parameter using an AGREE statement, such as the following:

Listing 1: AGREE example of modeling logical behavior.

```
assign altitude = if(heading < 180)
    7,000
else
    8,000;
```

The scientific model estimation (A-3) projects the value of a parameter based on the given equation that models the continuous dynamics. The estimation process also uses a time parameter, which monotonically increases with each step of execution. For example, a module that tracks distance may do so based on an assumption of a fixed rate of increase in one direction over time. As the time parameter increases during execution, the distance variable is calculated as a function of rate and time. The calculation is treated as an estimation since it does not necessarily account for all factors. As one of the contributions of this article, a tool has been created to perform linearization for AGREE representations, which can be referenced from component models (A-4). Such an AGREE expression is shown as follows:

Listing 2: AGREE example of modeling scientific model estimation.

```
eq t: real = 0.0 -> pre(t) + 1.0;
assign distanceEstimate = getDistance(t,
rate);
```

where "getDistance" is an AGREE function generated by translating the equation $d = \text{rate} * t$.

1) Linear Approximation: A module relies on measurements it receives from the environment to calculate KPPs, via the input/output (I/O) module. To calculate the system trajectory and predict the future state, modules must apply scientific theorems and perform potentially complex calculations. However, AADL/AGREE does not support complex mathematics; thus, there is the need for linearized equations for approximation.

The linearization y_L of a function y involves building a piecewise function, where each branch is of the form $mx + b$, and that the difference between the original and linearized function is minimized. For example, the function

$$y = 20x^2 - 150x + 500$$

becomes

$$y_L = \begin{cases} -102x + 471.2, & x \leq 2.4 \\ -6x + 240.8, & 2.4 < x \leq 4.8 \\ 90x - 220, & 4.8 < x \leq 7.2 \\ 186x - 911.2, & 7.2 < x \leq 9.6 \\ 282x - 1,832, & x > 9.6. \end{cases}$$

Algorithm 1 shows the steps to create a piecewise approximation of any function on a certain range if the function is differentiable and separable on that range. This is the classical bounded variable piecewise linearization algorithm, as seen in [23], and is implemented as a Python program.

Once the above aspects are modeled, the next step in component modeling is to create the AGREE assumptions and guarantees (A-5). The assumptions and guarantees use variables defined in the component model to represent the objective. The following example uses variables defined by the above snippets:

Listing 3: AGREE example of an assumption and a guarantee.

```
assume "Heading is within bounds":
    heading >= 0 and heading < 360;
guarantee "Altitude is kept above 500 ft":
    altitude > 500 ;
```

Algorithm 1: Generate Piecewise Approximation Function $G = \{g_k(x)|x \in R_k\}$, Where R_k is the Subrange and $g_k(x)$ is the Linear Function for the Given Range R = $[x_{stop}, x_{start}]$ with C as Total the Number of Cases and k Being an Integer from 1 to C.

Input: $f(x), R, C$;
Output: $G = \{R_k \rightarrow g_k(x)\}$;
$\delta \leftarrow ((x_{start} - x_{stop})/C)$;
for *i = 0 until C* **do**
  $x_i \leftarrow x_{start} + i * \delta$;
  $x_{midpoint} \leftarrow x_{start} + i * \delta + \delta/2$;
  $g_k(x) = f'(x_{midpoint}) * x + f(x_{midpoint})$;
  $G_k \leftarrow ([x_i, x_{i+1}] \rightarrow g_k)$;
return G

The objective can also be the identification of satisfactory values of the KPP. KPPs can be identified from the equations representing the kinematics or the logical behaviors. For example, in our case, we identified the KPP to be time to recovery. The value for the KPP was then represented within a guarantee statement for the component or whole system under consideration. Then, we performed assume–guarantee-based automated reasoning on the behavior (kinematics or logical behavior) of the component to evaluate whether a particular value for recovery time was satisfied. In order to compute the KPP under discussion, kinematic equations implemented within the component were evaluated. The evaluation was performed based on values of continuous parameters such as velocity, acceleration, and altitude. The value in the guarantee was then checked to see if it matched the computed value from the equation. The computed value was evaluated in an iterative process, by changing the value of the recovery time within the guarantee statement until the guarantee was satisfied, i.e., the value for the KPP matched what was computed from the equation.

Once the AGs are written, each submodule will have two associated AADL objects: a component type and a component implementation (A-6).

D. Verification

In this assume–guarantee-based approach, we verify a component (step 5) or a full system (step 7) by similar means. In either case, verification entails launching a model checker (step B-1). A model checker generates all the possible execution traces for the model under verification, to check if the property is satisfied or not. The result of model checking is either a "pass" or a "fail" with given counterexamples. A counterexample is one such trace generated by the model checker that disproves the property. Analyzing counterexamples (B-2) is required to identify the reason for failure and aids in changing the design to create a model that passes verification. Verification failures are due to different types of inconsistencies in the model. Since the model checker is applying formal verification, false positives are very uncommon (false positive meaning erroneous successful verification); however, false negatives are common. Removing true-negative inconsistencies also removes failure potentials from the design and, subsequently, the implementation.

In removing inconsistencies (B-3), we must constrain the model checker, so that it does not provide nonrealistic counterexamples. Two common false negatives arise from too loose assumptions and too strict guarantees, neither of which reflect reality. In the case of too loose assumptions, the model checker finds a counterexample that is an impossible situation, e.g., altitudes below ground level; we can assume that airplanes are flying above ground. In the case of too strict guarantees, we ask too much from the system, e.g., escaping harms way instantaneously, which is not possible.

1) Formal Compositional Verification: Once the system requirements are represented in an architectural framework, assume–guarantee-based structural reasoning can be applied to first verify the behavior of each of the components and then that of the composition of the overall system. We illustrate the described approach in our implementation in AADL/AGREE. In our approach, each of the systems is represented as shown in Fig. 3.

A system ($SystemN$) is defined by specifying the assumptions for its inputs (data or environmental), ($SystemN_A$), such as the normal operating ranges, velocity, and acceleration. In our case study, the velocity was assumed to be 50 ft/s and acceleration 100 ft/s^2. The expected guarantees for the outputs of the system are represented by ($SystemN_G$), such as time to recovery from three emerging situations occurring sequentially. The guarantee is a verification query, which initiates the execution of compositional reasoning to discover the performance boundary. For example, in our case study, the query states: "Is it possible that the time to recovery for the overall system is within 15 seconds?" The assumptions and guarantees of the components embedded within the system ($SystemN_{Arch(A,G)}$) are explored to check if the guarantees of the overall systems can be satisfied. In our example, the guarantees from each of the components for time to recovery are identified to be 3.7 s for AutoGCAS, 9.1 s for automated traffic collision avoidance system (AutoTCAS), and 1.7 s for automated geo-fence controller (AutoGFC) for the assumed velocity and acceleration. Therefore, the guarantees from each of the subsystems are composed together, to evaluate if the system-level guarantee ($SystemN_G$) is met. For our example, the total time to recovery for the composition of components is computed to be 3.7+9.1+1.7=14.5. The computed total time to recovery is less than 15 s, so the system-level guarantee is met. As part of the approach, we need to identify inputs, outputs, the internal architecture, and the assumptions and guarantees associated with each of the systems and its components. Once we have a representation for the whole system, we will be able to verify the correctness of the behavior by performing compositional reasoning.

E. Implementation

Once the whole system is verified, the next step is to automatically generate source code. The method of automatic source code generation (Step 8) is executed as shown in Section V.

The source code produced is intended to be a software "skeleton," meaning it is not yet a complete program with the desired behavior. Rather, each assertion becomes a piece of the structure, which automatically performs I/O checking based on the assumptions and guarantees modeled in an architectural framework such as AGs in the AADL/AGREE model. The source code has explicit places, in fact, where further implementations are required. Details that are not considered during model checking are now needed for machine instruction (step 9). The implementations for each module are imported or developed at this point. The output is a program with code specifying an application programming interface (API) and performing I/O validation, which is generated by the models, alongside the implementation desired to be integrated into the safety-critical system.

IV. DMC Design

The DMC is the core of the decision-making architecture. It selects the correct controller based on state and environmental conditions To this end, the DMC performs four functions: checking that a solution exists, building static maneuver schedules, relaying active maneuvers to the I/O module, and managing the active module. Real-time maneuver scheduling is done via the parameters that accompany the maneuvers received by the DMC, which help determine the time window in which the maneuver must take place. When a maneuver reaches the current moment in time, the DMC forwards the control from a module to the I/O module, which then forwards the control on to the desired output devices.

A. Maneuver Scheduling

The DMC generates a schedule of maneuvers depending upon the emerging scenario. Maneuvers not yet initiated can be rescheduled by the DMC to improve performance or deal with safety concerns. We assume nonpremptive scheduling so that no maneuver is stopped or updated once it is initiated. The steps to update the schedule are given by Algorithm 2. Algorithm 2 is a traditional implementation of an earliest deadline first scheduling algorithm without preemption and is described in [23]. It ensures that maneuvers are scheduled without overlap in order of closest time to failure. This serves as the most general-purpose strategy, that is, it may not always produce the best schedule in every situation.

B. Concurrency Analysis

One beneficial improvement to the scheduling is to allow for concurrent maneuvers if possible. There are at least two considerations in allowing concurrency: maneuver compatibility and resource limitations. Performing maneuvers concurrently increases the possible application space to complex systems, whose behavior includes a diverse set of behavior periods ranging from very short to very long, and systems with overlapping maneuvers. It also increases the possible performance of the decision-making architecture.

Algorithm 2: Create or Update Asynchronous Schedule $S_n = \{\gamma_0, \gamma_1, ...\gamma_i\}$ to S_{n+1} Where Event $\gamma = (m, t_{start})$, Maneuver $m = (C, t_{recover}, t_{failure})$, and Controls $C = \{c_0, c_1, ..., c_i\}$ Rescheduled Given Maneuver m_{new} at Time $t_{current}$.

Input: $S_n, m_{new}, t_{current}$;
Output: S_{n+1};
$M_{upcoming} \leftarrow \{\}$;
for γ *in* S_n **do**
  if $T_{start}(\gamma) > t_{current}$ **then**
    $M_{upcoming} \leftarrow M_{upcoming} \cup M(\gamma)$;
$S_{n+1} \leftarrow \{\}$;
$t_{next\_free} \leftarrow t_{current}$;
while $|M_{upcoming}| > 1$ **do**
  $m_{next} \leftarrow \underset{t_{failure}}{\operatorname{argmin}}\{m \in M_{upcoming}\}$;
  $M_{upcoming} \leftarrow M_{upcoming} - m_{next}$;
  $S_{n+1} \leftarrow S_{n+1} \cup (m_{next}, t_{next\_free})$;
  $t_{next\_free} \leftarrow t_{next\_free} + T_{recover}(m_{next})$;
return S_{n+1}

In the world of general-purpose computing, parallel processing is a heavily researched and well-understood topic. Implementing a job queue distributes computational load across multiple processes so that programs can scale according to available resources. In this framework, the maneuvers can function as jobs in a queue, assigned by the DMC to output devices (workers). Thus, we can apply theorems from distributed computing.

By design, the DMC determines the "concurrency factor" for a given schedule. The "concurrency factor" indicates the maximum number of maneuvers done concurrently at a single moment, if the amount of concurrency is minimized. In other words, it corresponds to the minimum number of workers needed to perform a number of jobs within a certain amount of time. Determining the concurrency factor allows the DMC to make guarantees about concurrency or lack thereof during the verification stage.

C. Generating Schedule Validator

When multiple emerging situations occur, the DMC needs to exhibit the behavior of a schedule validator. To calculate the concurrency factor, a different function is needed for all possible maneuvers. Presumably, this would require manually writing the needed code for every needed function. This article contributes an automated process for generating the needed functions for determining the concurrency factor. The process generates a concurrency factor evaluation that is implemented in the AADL/AGREE framework, to show the feasibility of the approach. Listing 4 shows a generated AGREE function for three maneuvers.

Listing 4: This AGREE node was generated.

```
node concurrencyFactor(t0: real, t1: real,
                t2: real, totalTime: real)
 returns(can_use_1: bool, can_use_2: bool,
                can_use_3: bool);
let
    can_use_1 = ((t0 + t1 + t2)
    < totalTime);
    can_use_2 = ((t0) < totalTime
    and (t1 + t2) < totalTime)
    or ((t0 + t1) < totalTime and (t2)
    < totalTime)
    or ((t1) < totalTime and (t0 + t2)
    < totalTime);
    can_use_3 = ((t0) < totalTime
    and (t1) < totalTime
    and (t2) < totalTime);

tel;
```

The generated function returns n Boolean flags that correspond to the truth of the generic statement: "can the given maneuvers be performed with concurrency factor n."

To calculate the concurrency factor, the generated function checks all distinct arrangements of the given maneuvers to determine which arrangements satisfy the time constraint. In other terms, valid$(S(M,n,t)) \equiv (A_1 \vee A_2 \vee \cdots \vee A_k)$, where S is a schedule of M maneuvers, and A is a valid arrangement of the maneuver set M, where the last maneuver finishes before time t. An arrangement is successful if the sum of time-to-recovery values for each worker is less than the total time. Furthermore, valid$(A(M,n,t)) \equiv (\text{sum}(W_0) < t) \wedge (\text{sum}(W_1) < t) \wedge \cdots \wedge (\text{sum}(W_i) < t)$, where W_i indicates the list of maneuvers performed by worker i.

Algorithm 3 gives the whole process of calculating the concurrency factor. The underlying function allPartitiong() gives all distinct groupings of a list of objects. For example

$$allPartitionings([A,B,C]) = [[A,B,C]],[[A],[B,C]],[[A,B],[C]],[[A],[B],[C]].$$

The result is a list of four partitions of the given three elements. The ordering of the list must be preserved when initially partitioning, but the execution order of a partition (and elements within a partition) can be ignored. Therefore, the collection $[A,B]$ is equivalent to $[B,A]$ and $[A],[B,C]$ is equivalent to $[C,B],[A]$. Applying this to the jobs and workers scenario, the first partitioning uses one worker, the second two use two workers, and the last partitioning uses three workers. This function gives the complete list of valid and distinct partitionings for determining the concurrency factor (analogous to the minimum number of workers required).

Algorithm 3: AGREE Concurrency Factor Node given the Maneuver Set M, the Number of Maneuvers to Schedule n, and t the Amount of Time.

```
1  Input: M, n, t ;
2  Output: C ;
3  C ← {} ;
4  for partitioning ∈ allPartitioning(M) do
5      numOfGroups ← len(partitioning);
6      if numOfGroups ≤ n then
7          for group ∈ partitioning do
8              if sum(group) ≤ t then
9                  C[numOfGroups] ← true ;
10 return C
```

V. Code Generation

The process of creating a functional solution that fully adheres to a verified design poses challenges. The difficulty is further increased if there are several instances of behavior, which diverge from design, as it has the potential to create unforeseen and unverified side effects that should absolutely be avoided in safety-critical systems. The effort in verifying design is lost if the implementation exhibits such divergent behavior. We present a process for generating compliant implementations expressed in a modern high-level programming language.

A. Translation Process

Producing a compliant implementation involves automatically generating verification-enhancing code with an API for further extension. In the following, we present a mapping of AADL to Java, which is applied during code generation.

The code generation from AADL to Java is formally expressed in Algorithm 4. It creates Java class representations of each AADL module. Each Java class performs I/O verification that exactly matches the assume/guarantee expressions in AGREE. The resultant code uses object inheritance to express the design abstractions as object abstractions, such that implementations are easily added following code generation. The implementation is bounded by the constraints defined in AADL as expressed in Java after translation. Once the full implementations are written, it can be added in conjunction with the generated code in the system's codebase for further development.

The process begins with the AADL packages, creating a set of Java classes to capture these higher level groupings. Inside each AADL package is a number of component types and implementations, where the component types are captured by Java classes. The implementation of the component behaviors is ignored by the code generation process, since it is replaced by manually written implementations later on by developers, but the verified constraints form the bounding specifications for the implementation and are generated. The generation process also includes a small library of classes to perform basic actions needed by all uses of this tool. The result is a set of Java classes,

each expressing pieces of the corresponding AADL/AGREE framework.

The translation process can be broken roughly into two types: translating component behavior and translating AGREE nodes. AGREE annexes for component models might invoke AGREE nodes; thus, there is the need for translating AGREE nodes. An AGREE node is similar to a library function in the standard programming practice. AGREE nodes may have multiple return values, requiring special return objects in Java. Therefore, each AGREE node has an associated type class. AGREE nodes are also combined by package into enclosing Java classes.

The main type of translation, translating component behavior, involves matching AADL component type and AADL component implementation definitions to create corresponding classes for each pair that creates the validation layer. In this layer, AGREE assumptions and guarantees are modeled as assert statements in Java, which create Java runtime exceptions if the required conditions are not met. Assumptions are asserted on values before the implementation call is made; guarantees are asserted after an implementation call is made. Asserts are used as a direct semantic mapping, which does not follow ideal coding principles, but allows preservation of constraint in the automated mapping process. Ideally, we would throw exceptions that provide explanation for the issue. The implementation call is a one line call to the stub (i.e., empty but prepared for extension) method. Each component pair has an associated abstract class, which contains only this abstract method stub.

B. Code Generation Algorithm

The algorithm begins with each given AADL package (p) and every component (a) in those packages, which lead to the generation of Java code that will perform consistency checks according to the model. Each package is assumed to have pairs of components, which refer ultimately to the same software product, one defining the type and one restricting the implementation. Those pairs are both considered when generating a single representative Java class (c). The AGREE annex (R) contains the necessary assumptions (A), guarantees (G), and expressions (E). Assumptions are translated into input validation (V_{in}), ensuring that the environment and given data sources align with specified AGREE assumptions. Guarantees are translated into output validation (V_{out}), ensuring that the complex algorithm produces compliant data. After input validation and before output validation, μ_{call} represents a method call to the complex algorithm. That complex algorithm will use the given inputs, accessible as class fields, and provide outputs, also as class fields. Those class fields are checked against translated assumptions and guarantees to ensure consistency during the input and output validation steps, respectively.

The input to this algorithm (line 1) is primarily the AADL packages P defining the architecture. A utility class library c_U is also required to support the translated product. For example, the utility class implements the "pre" function from AGREE. The output of this algorithm (line 3) is a set of Java classes C, which fulfills the need for validation.

Algorithm 4: Generate Java Class Objects from AADL Objects, Each with AGREE Annexes.

1 **Input**: $P : \{p\}, c_U$;
2 **Where**: $p : (\{n\}, \{(a_{type}, a_{impl})\})$;
3 **Output**: $C : \{c\}$;
4 **Where**: $c : (F, M)$;
5 $C \leftarrow \{c_U\}$;
6 **for** *each package* $p \in P$ **do**
7 **for** *each pair* $(a_{type}, a_{impl}) \in p$ **do**
8 $I_{\cup} \leftarrow a_{type} \cup a_{impl}$;
9 $I_{\cup} : (S, \Phi, R)$;
10 $R : (\{A\}, \{G\}, \{E\})$;
11 $F \leftarrow T_{J \leftarrow vars}(R, \Phi)$;
12 $T_{exp} \leftarrow T_{J \leftarrow exp}(E)$;
13 $V_{in} \leftarrow T_{J \leftarrow assert}(A)$;
14 $V_{out} \leftarrow T_{J \leftarrow assert}(G)$;
15 $U \leftarrow \{\}$;
16 **for** *each port* $\phi_{out} \in \Phi$ **do**
17 $\phi_{out} : (a_{adjacent}, \phi_{in})$;
18 $U \leftarrow U \cup T_{update}(\phi_{out} \mapsto \phi_{in}, a_{adjacent})$;
19 $m_{impl} \leftarrow m_{abstract} : (a_{name})$;
20 $m_{validation} \leftarrow m : (V_{in}, T_{exp}, \mu_{call}(m_{impl}), V_{out}, U)$;
21 $M \leftarrow \{m_{impl}, m_{validation}\}$;
22 $c_v \leftarrow c : (a_{name}, F, M)$;
23 $C \leftarrow C \cup c_v$;
24 $M_{nodes} \leftarrow \{\}$;
25 **for** *each AGREE node* $n \in p$ **do**
26 $m_n \leftarrow m : (n_{name}, T_{J \leftarrow exp}(n))$;
27 $M_{nodes} \leftarrow m_{nodes} \cup m_n$;
28 $C \leftarrow C \cup T_{type}(n)$;
29 $c_{nodes} \leftarrow c : (p_{name}, m_{nodes})$;
30 $C \leftarrow C \cup c_{nodes}$;
31 **return** C

The outer for-loop (line 6) iterates through each given AADL package p. AADL packages outline a scope and namespace for components and AGREE nodes, with all constructs belonging to exactly one package. The for-loop on line 7 examines each AADL component pair: one component type a_{type} and its associated component implementation a_{impl}. Lines 8–14 create the pieces needed by the Java method to perform validation.

Lines 15–18 create update statements U to translate the AADL port construct ϕ. In AADL, a port allows sharing of information between components. Therefore, in the Java program, the functionality of a port to share information between classes is performed via updates. That is, one class sets the value of a member field of another class with statements after output validation has succeeded. To generate the update statements, we iterate through port connections to extract the necessary parameters, e.g., source and destination.

Lines 19–23 assemble the class c_v resulting from the pair of AADL component objects. The class contains the necessary member fields F for data accessibility, a method for validation $m_{\text{validation}}$, and an abstract method for API definition m_{abstract}.

Lines 24–28 translates each AGREE node n defined in the current package. Each AADL package results in one Java class c_{nodes} containing the AGREE nodes M_{nodes} that were contained inside that AADL package. The AGREE nodes are represented as methods m_n, which take as arguments the same arguments as expressed in AGREE. They return, however, a composite object since Java only supports a single return type. That composite object is a plain Java type class containing only the member fields that map to the return types of the AGREE node.

Lines 29 and 30 assemble the AGREE-node-based classes with the AADL-component-based classes into one set of classes in preparation for the next AADL package iteration or final return.

This translation process supports the systems/software engineering life cycle, by automating the translation of requirements/constraints modeled during the design phase to the source code implemented during the construction phase. It enables early analysis of requirements to identify any conflicts during the design phase and also emphasizes reducing requirement/constraint mismatch between the source code and the architectural design.

VI. Case Study: Autonomous Aerial System

Verification of real-time systems is needed in order to develop autonomous aviation systems, which must be trusted and resilient. For our case study, the system will invariably become complex as it becomes resilient, since it must cope with multiple midair possibilities while remaining safe. There is a large potential for damage in the form of property loss and injury during an aviation system failure. This article emphasizes the need to apply formal method-based assurance methodologies during design time.

A. Selecting Autonomous Aviation Subsystems

In developing a case study, we use three existing aviation subsystems that can benefit from our techniques. They represent functionalities that usually were performed by humans or benefit from human decision making to be resilient. We then apply our verification process to a system composed of all three subsystems to show that compositional verification is achievable in this instance. First, we provide a description of the three autonomous subsystems, which are under examination.

1) Traffic Collision Avoidance System: The TCAS [24] subsystem attempts to keep a safe distance between all aircraft equipped with this system. Nonautonomous TCAS recommends safe altitudes to the pilot through a cockpit display. Often, the pilot also communicates with the local air traffic controller to receive and confirm altitude adjustments. Some effort is underway to make TCAS autonomous; in fact, Airbus has deployed a partially autonomous TCAS (still using input from the pilot) that simplifies the steps to maneuver safely [25].

2) Ground Collision Avoidance System (GCAS): The GCAS' [26] responsibility is to prevent collision with the ground. Standard GCAS uses audiovisual cues to prompt and warn the pilot if a ground collision is detected. AutoGCAS has been deployed in military aircraft [27].

```
assume "Maximum Descent Velocity is not less than -10.0":
    maximumDescentVelocity >= -10.0;

assume "Aircraft starts above minAltitudeAllowed":
    t = 0.0 => altitude > minAltitudeAllowed;

guarantee "Aircraft is climbing and above min altitude":
    t >= gcasRecoveryTime => (altitudeAfterClimb > minAltitudeAllowed)
        and (altitudeAfterClimb > pre(altitudeAfterClimb));

guarantee "Aircraft has not broken altitude constraint":
    t < gcasFailureTime => (altitudeAfterDescent > failureAltitude);

guarantee "Report time-to-recovery": timeToRecovery = gcasRecoveryTime;
guarantee "Report time-to-failure": timeToFailure = gcasFailureTime;
```

```
assign altitudeAfterClimb = altitude ->
    gcas_nodes.aircraftClimbAltitude(t);

assign altitudeAfterDescent = altitude ->
    gcas_nodes.aircraftDescentAltitude(t);
```

Fig. 4. GCAS AADL/AGREE model snippets.

3) Geo-Fence Controller (GFC): A Geo-Fence [28] is a geometric three-dimensional boundary that an aircraft must not cross. A Geo-Fence can be used to create volumes/zones of prohibitive entry or, in an optimistic sense, an "operational area" [28]. An AutoGFC, then, would prevent a vehicle from crossing this boundary via changing course or a simple halt.

In this article, we focus on explaining the GCAS model and its inner workings guided by assume–guarantee-based contracts. Similar models have been developed for TCAS and GFC.

B. Examining the GCAS Model

With three subsystems selected, we create models to verify internal behavior before verifying conjoined behavior. Fig. 4 shows most of the AutoGCAS models we have created as an example of a subsystem model; we leave out basic initialization code. The primary guarantee for AutoGCAS is that the aircraft shall never go below a certain minimum altitude. In the model, this is captured by the four guarantee statements. The first dictates that the subject is safe after the time to recovery has elapsed, given that a recovery was initiated at $t = 0$. The second guarantee dictates that the aircraft does not break the altitude constraint before the time to failure has elapsed. The third and fourth guarantees allow AADL to communicate the KPPs with other objects, namely, the DMC.

This model also relies upon two assumptions, the first of which is a bound on the descent velocity. This is necessary since an unbounded descent velocity would allow for instantaneous unstoppable failures. The selected value of -10.0 units is an example of a preconfigured parameter, unlike the altitude which changes based on the evolution of the system. The second assumption is that the aircraft starts above a prespecified minimum allowed altitude.

The implementation of "AutoGCAS.impl" is shown in Fig. 4, where gcas_nodes.aircraftClimbAltitude() and gcas_nodes.aircraftDescentAltitude() refers to the two estimator functions used by this component model. The first uses preconfigured knowledge about the climbing capability to determine the altitude at time t (given that a maneuver began at $t = 0$). Likewise, the second uses preconfigured knowledge pertaining to the maximum possible descent speed to determine the altitude

```
node kinematicLinearization(t: real) returns(y: real);
let
    y = if t <= 0.00 then -126.00*t + 492.80
        else if t > 0.00 and t <= 1.20 then -126.00*t + 492.80
        else if t > 1.20 and t <= 2.40 then -78.00*t + 435.20
        else if t > 2.40 and t <= 3.60 then -30.00*t + 320.00
        else if t > 3.60 and t <= 4.80 then 18.00*t + 147.20
        else if t > 4.80 and t <= 6.00 then 66.00*t + -83.20
        else if t > 6.00 and t <= 7.20 then 114.00*t + -371.20
        else if t > 7.20 and t <= 8.40 then 162.00*t + -716.80
        else if t > 8.40 and t <= 9.60 then 210.00*t + -1120.00
        else if t > 9.60 and t <= 10.80 then 258.00*t + -1580.80
        else 306.00*t + -2099.20;
tel;
```

Fig. 5. Linear approximation for GCAS kinematics.

at time t assuming a maximum descent began at time $t = 0$. Our approach generated both functions from aircraft kinematic models, in the AADL/AGREE syntax, in order to create these estimations. For example, the altitude is determined by the equation $\text{Alt} = 0.5\text{A}{*}t^2 + V_{\text{initial}}{*}t + \text{alt}_{\text{initial}}$. The approximation function is shown in Fig. 5.

With our models, we show that the time to recovery is 3.7 s for AutoGCAS, 9.1 s for AutoTCAS, and 1.7 s for AutoGFC.

C. Composing Models and DMC Design

With the three subsystem models written and analyzed, we are ready to combine them into one model. A top-level AADL system is created where the complex algorithm models—AutoGCAS, AutoTCAS, and AutoGFC—are designated as subsystems. In addition, we add a model describing the *DMC* to this system for representing the decision making between maneuvers (see Section IV for the DMC design).

We embed the *AutoGCAS*, *AutoTCAS*, and *AutoGFC* subcomponents in an AADL object we call the *supervised system* (defined in Fig. 7 and illustrated in Fig. 6), where top-level assumptions and guarantees can be verified. For instance, we found that all three maneuvers could complete in 15 s through compositional reasoning, whereby the guarantee of the top-level component is verified based on the guarantees provided by the subsystem-level guarantees composed together. More interestingly, we can prove the degree of concurrency required for a given window of time to failure. We can achieve scalability by deploying this method, as once the guarantee of each component is verified individually, we can consider those guaranteed outputs to verify the top level guarantees.

The AADL/AGREE snippet (see Fig. 6) and diagram (see Fig. 7) show us how the subsystems are connected to the DMC via KPPs for proving top-level guarantees. At this level, we see the guarantees for degrees of concurrency that can be proved.

Like the estimators used by the component models, the DMC determines the level of concurrency required for any combination of maneuvers using functions already generated (again in the AADL/AGREE syntax) by our solution to mitigate contingency scenarios (see Fig. 8). We generate these functions because they follow a pattern and can grow large beyond four components.

The DMC uses the concurrencyFactor() function to determine the degree to which concurrency is needed given a maneuver set and a deadline t. The function has n outputs of 1 to n, all Boolean variables corresponding to degree of concurrency possible. The nth output variable is "true" if the maneuver set can

```
system Supervised_System
    features
        totalTime: in data port real;

        timeToFailure: out data port real;
        timeToRecovery: out data port real;

        concurrencyDegree1: out data port bool;
        concurrencyDegree2: out data port bool;
        concurrencyDegree3: out data port bool;

        annex agree {**
            guarantee "Can complete sequentially": concurrencyDegree1;
            guarantee "Performable in two processes": concurrencyDegree2;
            guarantee "Performable in three processes": concurrencyDegree3;
        **};

end Supervised_System;

system implementation Supervised_System.impl

    subcomponents
        gcas: system AutoGCAS;
        tcas: system AutoTCAS;
        gfc: system AutoGFC;

        supervisor: system Supervisor;

    connections
        gcas_ttr: port gcas.timeToRecovery -> Supervisor.gcas_ttr   ;
        tcas_ttr: port tcas.timeToRecovery -> Supervisor.tcas_ttr   ;
        gfc_ttr: port gfc.timeToRecovery -> Supervisor.gfc_ttr      ;

        gcas_ttf: port gcas.timeToFailure -> Supervisor.gcas_timeToFailure  ;
        tcas_ttf: port tcas.timeToFailure -> Supervisor.tcas_timeToFailure  ;
        gfc_ttf: port gfc.timeToFailure -> Supervisor.gfc_timeToFailure     ;

        c1_conn: port supervisor.c1 -> concurrencyDegree1   ;
        c2_conn: port supervisor.c1 -> concurrencyDegree1   ;
        c3_conn: port supervisor.c1 -> concurrencyDegree1   ;

end Supervised_System.impl;
```

Fig. 6. Top-level supervised system AADL/AGREE model definition.

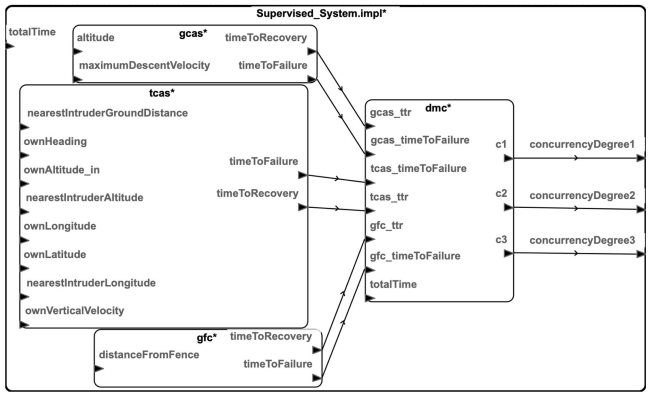


Fig. 7. Supervised system model diagram.

be performed given the deadline and no more than n maneuvers execute concurrently at any time.

D. Generated Code From Model

To demonstrate the translation from AADL to Java, the architecture model containing AutoGCAS, AutoTCAS, and AutoGFC has been translated into Java. Table I shows the GFC as an AADL model with the corresponding representation, along with a verification method in Java as generated by the tool. Once all models are represented in code and combined into a single software package, the complex algorithms are ready for integration.

The generated class contains the method checked_GeoFence_singleton() and the method unchecked_

```
eq concurrencyDegree1: bool,
    concurrencyDegree2: bool,
    concurrencyDegree3: bool =
        composite_nodes.concurrencyFactor(gcas_timeToRecovery,
                                          tcas_timeToRecovery,
                                          gfc_timeToRecovery,
                                          totalTime);

guarantee "Can complete sequentially": concurrencyDegree1;
guarantee "Performable in two processes": concurrencyDegree2;
guarantee "Performable in three processes": concurrencyDegree3;
```

Fig. 8. Snippet of the DMC model.

TABLE I
AADL TO JAVA TRANSLATION EXAMPLE

AGREE/AADL Model	Generated Java
system Geofence	**void** checked_GeoFence() {...}
eq t: real = 0.0 −> pre(t) + 0.1;	t.set(0.0, AadlUtil.pre(t) + 0.1);
assume "above ground": distanceFromGround > 0;	**assert** distanceFromGround.get() > 0
system Geofence.impl	call_GeoFence_impl();
guarantee "recovery occurs after some time": t >= timeToRecovery => distanceFromFence > pre(distanceFromFence);	**assert** (!((t.get()) .equals(timeToRecovery.get())) || ((distanceFromFence.get() > AadlUtil.pre(distanceFromFence))));
guarantee "ttr is 1.7": timeToRecovery = 1.7;	**assert** timeToRecovery.get() .equals(1.7);
guarantee "ttf is 2.0": timeToFailure = 2.0;	**assert** timeToFailure.get() .equals(2.0);
timeToRecovery −> DMC.geof_timeToRecovery	DMC.geof_timeToRecovery .set(timeToRecovery.get());
timeToFailure −> DMC.geof_timeToFailure	DMC.geof_timeToFailure .set(timeToFailure.get());

TABLE II
AADL PORT TRANSLATION

AADL port ↦ Java Member field
ownLongitude: in data port real ↦ static AadlMemoryObject<Double> ownLongitude
ownLatitude: in data port real ↦ AadlMemoryObject<Double> ownLatitude
timeToRecovery: out data port real ↦ AadlMemoryObject<Double> timeToRecovery
timeToFailure: out data port real ↦ AadlMemoryObject<Double> timeToFailure

GeoFence_singleton(); the latter method is called by the former in the second statement in Table I. The behavior of the experimental controller belongs to the *unchecked* method; it is called by the *checked* method with validation to create a guarded experimental module.

Another concept to capture in code is the sharing of information between AADL objects via ports. To accomplish this, the Java representation uses public class member fields, as shown in Table II.

The program developed to generate Java representations such as in Table II performs different levels of parsing and semantic mappings to accomplish its task. The tool uses syntactic and structural knowledge of both AADL and Java to extract concepts from AADL and appropriately translate them into Java. Two low-level syntactic mappings used are the following.

1) *Implies statements:* AADL accepts the syntax $A => B$ to mean "A implies B," which has no direct equivalent in Java. The logical identity $(A \rightarrow B) \iff (!A \vee B)$ allows us to represent such statements in Java: `!A || B`. Parenthesis are then added to ensure correct order of operations.
2) *If-expression:* While Java obviously contains the if-statement, it is not equivalent to the if-expression present in AGREE (and other languages), e.g., `R = if (C) then T else F`. The Java ternary operator, however, is equivalent: `R = C ? T : F`.

The code generation tool helps bridge the gap between design and implementation and lessens the burden on manual software development. Having such a tool also makes the process faster without sacrificing correctness. By translating AGREE assumptions and guarantees, we can create valid assertions to enforce at runtime, expressed in the development language.

VII. CONCLUSION

The research discussed in this article proposed a framework that can be used in the integration of complex algorithms. The methodology augments existing development processes, adding rigorous design verification to ensure consistency and correctness early in the process.

In the proposed approach, we performed modeling and analysis of complex algorithms, which involved the identification of the logical behavior and the continuous dynamics associated with the algorithm. We showed how several complex algorithms can be architecturally integrated to perform compositional reasoning to guarantee the overall performance of the system. We also discussed the importance in the design of the DMC and the analysis of the DMC behavior in relation to the identification of the existence of a solution, executed sequentially or concurrently. We also demonstrated automated translation of the architecture to the outline of the source code, which preserves the constraints that were modeled in the architecture to reduce the gap between design and implementation. The future work will involve generation of runtime monitors to alert other systems or personnel if these constraint boundaries are violated.

All codes developed in this article can be found at the author's GitHub page, in the Applied Modeling project [29]. Code generation is found in [30], and the AADL models are found in [31].

REFERENCES

[1] N. Hovakimyan and C. Cao, *L1 Adaptive Control Theory: Guaranteed Robustness with Fast Adaptation* (ser. Advances in Design and Control). Cambridge, U.K.: Cambridge Univ. Press, 2010.
[2] D. Jourdan, M. Piedmont, V. Gavrilets, D. Vos, and J. McCormick, "Enhancing UAV survivability through damage tolerant control," in *Proc. AIAA Guid., Navigat. Control Conf.*, 2010, Art. no. AIAA 2010-7548.
[3] K. Wise, E. Lavretsky, and N. Hovakimyan, "Adaptive control of flight: Theory, applications, and open problems," in *Proc. Amer. Control Conf.*, 2006, pp. 5966–5971, doi: 10.1109/ACC.2006.1657677.
[4] P. Abbeel, A. Coates, M. Quigley, and A. Y. Ng, "An application of reinforcement learning to aerobatic helicopter flight," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2007, pp. 1–8.
[5] M. Bertozzi, A. Broggi, M. Cellario, A. Fascioli, P. Lombardi, and M. Porta, "Artificial vision in road vehicles," *Proc. IEEE*, vol. 90, no. 7, pp. 1258-1271, Jul. 2002.
[6] S. Bhattacharyya *et al.*, *Enhancing Autonomy with Trusted Cognitive Modeling*. Arlington, VA, USA: Association for Unmanned Vehicle Systems International, 2015.

[7] S. Chien, R. Sherwood, D. Tran, B. Cichy, and G. Rabideau, "The EO-1 autonomous science agent," in *Proc. Int. Joint Conf. Auton. Agents Multiagent Syst.*, 2004, pp. 420–427.

[8] D. Swihart *et al.*, "Automatic ground collision avoidance system design," *IEEE Aerosp. Electron. Syst. Mag.*, vol. 26, no. 5, pp. 4–11, May 2011.

[9] G. Hagen, R. Butler, and J. Maddalon, "Stratway: A modular approach to strategic conflict resolution," in *Proc. 11th AIAA Aviation Technol., Integr. Oper. Conf.*, 2011, Art. no. AIAA 2011-6892.

[10] CollinsAerospace, "Avoidance rerouter," [Online]. Available: https://www.collinsaerospace.com/en/what-we-do/Military-And-Defense/Avionics/Software-Applications/Avoidance-Re-Router-Arr-7000, Accessed: Sep. 20, 2020.

[11] S. Bhattacharyya, D. Cofer, D. Musliner, J. Mueller, and E. Engstrom, "Certification considerations for adaptive systems," in *Proc. Int. Conf. Unmanned Aircr. Syst.*, 2015, pp. 270–279.

[12] D. Phan *et al.*, "A component-based simplex architecture for high-Assurance cyber-physical systems," in *Proc. Int. Conf. Appl. Concurrency Syst.Des.*, Jun. 2017, pp. 49–58.

[13] S. Bak, D. K. Chivukula, O. Adekunle, M. Sun, M. Caccamo, and L. Sha, "The system-level simplex architecture for improved real-time embedded system safety," in *Proc. 15th IEEE Symp. Real-Time Embedded Technol. Appl.*, 2009, pp. 99–107.

[14] L. Sha, J. B. Goodenough, and B. Pollak, "Simplex Architecture : Meeting the Challenges of Using COTS in high-reliability systems," *CrossTalk*, pp. 7–10, 1998. [Online]. Available: http://www.sei.cmu.edu/technology/dynamic_systems/simplex/

[15] B. Templeton, "NTSB report on tesla autopilot accident shows what's inside and it's not pretty for FSD," 2019. [Online]. Available: https://www.forbes.com/sites/bradtempleton/2019/09/06/ntsb-report-on-tesla-autopilot-accident-shows-whats-inside-and-its-not-pretty-for-fsd/

[16] X. Wang, N. Hovakimyan, and L. Sha, "RSimplex," *ACM Trans. Cyber-Phys. Syst.*, vol. 2, no. 4, pp. 1–26, 2018.

[17] *System Model. Lang., sysML.* [Online]. Available: http://www.sysml.org, Accessed: Sep. 20, 2020.

[18] *Architecture Anal. and Des. Lang., aADL.* [Online]. Available: http://www.aadl.info/aadl/currentsite/, Accessed: Sep. 20, 2020.

[19] *Open Source Architecture Tool Environ.*. [Online]. Available: http://www.aadl.info/aadl/currentsite/tool/osate.html

[20] S. Bhattacharyya *et al.*, "Verification of quasi-synchronous systems with uppaal," in *Proc. IEEE/AIAA 33rd Digit. Avionics Syst. Conf.*, 2014, pp. 8A4-1–8A4-12.

[21] G. Behrmann, A. David, and K. G. Larsen, "A tutorial on uppaal," in *Formal Methods for the Design of Real-Time Systems* (Lecture Notes in Computer Science), vol. 3185. New York, NY, USA: Springer, 2004, pp. 200–236.

[22] D. Cofer *et al.*, "Secure mathematically-assured composition of control models," Air Force Res. Lab., Wright-Patterson Air Force Base, OH, USA, Tech. Rep. AFRL-RI-RS-TR-2017-176, 2017.

[23] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. Cambridge, MA, USA: MIT Press, 2009.

[24] J. K. Kuchar and A. C. Drumm, "The traffic alert and collision avoidance system," *Lincoln Lab. J.*, vol. 16, no. 2, pp. 277–296, 2007.

[25] J. Meyer, M. Göttken, C. Vernaleken, and S. Schärer, *Automatic Traffic Alert and Collision Avoidance System (TCAS) Onboard UAS*. Dordrecht, The Netherlands: Springer, 2015, pp. 1857–1871.

[26] A. Burns, D. Harper, A. F. Barfield, S. Whitcomb, and B. Jurusik, "Auto GCAS for analog flight control system," in *Proc. AIAA/IEEE Digit. Avionics Syst. Conf.*, 2011, pp. 1–11.

[27] L. Martin, "Saving the good guys: Eighth save illustrates life-saving auto GCAS technology." [Online]. Available: https://www.lockheedmartin.com/en-us/products/autogcas.html, Accessed: Sep. 20, 2020.

[28] K. J. Hayhurst, J. M. Maddalon, N. A. Neogi, and H. A. Verstynen, "A case study for assured containment," in *Proc. Int. Conf. Unmanned Aircr. Syst.*, 2015, pp. 260–269.

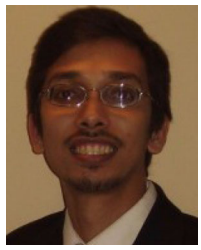
[29] M. Stafford, *All Code Repositories Related to This Res.*. [Online]. Available: https://github.com/users/miltoncs/projects/2, Accessed: Sep. 20, 2020.

[30] M. Stafford, *Code Gener. Code Repository on GitHub*. [Online]. Available: https://github.com/miltoncs/AutoModelCheckerGeneration, Accessed: Sep. 20, 2020.

[31] M. Stafford, *Aviation Models Repository on GitHub*. [Online]. Available: https://github.com/miltoncs/AviationRequirementModeling, Accessed: Sep. 20, 2020.

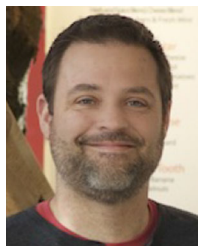
Milton Stafford received the bachelor's and master's degrees in computer science from the Florida Institute of Technology, Melbourne, FL, USA, in 2016 and 2019, respectively.

His thesis applied formal modeling techniques and source code generation in an effort to improve safety-critical system development. He is currently a Developer with Maxar Technologies, Westminster, CO, USA, and a part-time Researcher.



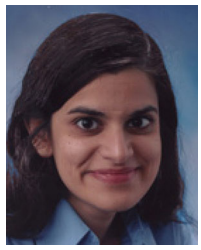
Siddhartha Bhattacharyya (Member, IEEE) received the bachelor's degree in electrical and electronics from BIT Mesra, Jharkhand, India, in 2001, the master's degree in electrical engineering from Iowa State University, Ames, IA, USA, in 2003, and the Ph.D. degree in electrical engineering, with a focus in formal methods and autonomous systems, from the University of Kentucky, Lexington, KY, USA, in 2005.

He is currently a Faculty Member with the Florida Institute of Technology, Melbourne, FL, USA. He was previously a Senior Research Engineer with Rockwell Collins. Advanced Technology Center, where he worked on research programs for assurance of safety-critical systems. He primarily conducts research in the area of formal methods for the design, verification, and validation of intelligent autonomous systems, avionics, cyber security, and systems biology.



Matthew Clark received the bachelor's degree in electrical engineering with a focus on electrical power and intelligent control systems and the master's degrees in electrical engineering from Wright State University, Dayton, OH, USA, in 2000 and 2009, respectively.

He is a Principal Scientist with Galois Inc., Dayton, OH, USA. He is also the Principal Investigator (PI) for the Defense Advanced Research Projects Agency (DARPA)'s Cyber Assured Systems Engineering integration technical area and a PI for the DARPA's Assured Autonomy Program, where he is leading an effort to symbolic execution-based verification of machine learning algorithms. Prior to his work at Galois, he was a civilian at the Air ForceResearch Laboratory, leading research in the areas of runtime assurance and autonomous verification.



Natasha Neogi received the M.Phil degree in engineering from the University of Cambridge, Cambridge, U.K., in 1997 and the Ph.D. degree in aerospace, aeronautical, and astronautical engineering from the Massachusetts Institute of Technology, Cambridge, MA, USA, in 2002.

She is currently a Researcher with the NASA Langley Research Center, Hampton, VA, USA. She was previously a Staff Member of the Office of the Chief Scientist, NASA Headquarters. Her research interests include verification and validation of software-intensive safety-critical infrastructure systems, as well as certification issues concerning airworthiness of unmanned aircraft systems.

Thomas C. Eskridge received the B.S. and M.S. degrees in computer science from Southern Illinois University, Carbondale, IL, USA, in 1986 and 1987, respectively, and the Ph.D. degree in philosophy from Binghamton University, Binghamton, NY, USA, in 2012.

He is currently an Associate Professor with the Florida Institute of Technology, Melbourne, FL, USA. He was previously a Research Scientist with the Florida Institute for Human and Machine Cognition, studying knowledge organization. He conducts research in multiagent systems for cybersecurity, visualization, teamwork, and ontologies and knowledge representation. His research focus is human-centered computing, using computer systems to amplify human performance.