The Assume Guarantee Reasoning Environment with Application to an Unmanned Helicopter ⋆

John Backes, Darren Cofer, and Isaac Amundson

Collins Aerospace, Minneapolis MN 55438

Abstract. The Assume Guarantee Reasoning Environment (AGREE) is a compositional analysis tool for systems modeled in the Architecture Analysis and Design Language (AADL). It is compositional in that it can be used to prove properties about each layer of the architecture using properties of its components, continuing down the model hierarchy. The compositional analysis is performed in terms of assume-guarantee contracts that are attached to each component. Assumptions describe the expectations that a component has about its environment, while guarantees describe bounds on the behavior provided by the component. AGREE uses k-induction model checking as its underlying analysis algorithm. In this paper we describe the AGREE annex for adding assume-guarantee behavior contracts to AADL models. We demonstrate the capabilities of the AGREE analysis tool by using it to verify key properties of the command authorization logic for an unmanned helicopter.

1 Introduction

Commercial and military aircraft rely on complex collections of distributed real-time embedded software. This software is critical to the safe operation of the aircraft, and so presents many challenges for the organizations that develop it.

Model-based design (MBD) tools are commonly used to implement avionics software functions, but system-level design tools for specifying the interactions of distributed components, resource allocation decisions, and communication mechanisms are less mature. This has made it difficult to apply formal methods effectively at the system level. One of our goals has been to create a system modeling methodology that would incorporate existing practices and artifacts and be compatible with tools and processes used in industry.

The work described in this paper directly addresses this goal by using modeling tools that accurately capture the system architecture and support the integration of formal analysis tools, as well as generation of flight software from the same model. This work deeply embeds formal verification into the design process to enable correct-by-construction development of systems that work the first time. The use of components with formally specified contracts, design patterns that provide formally guaranteed properties, and an architectural modeling

⋆ This work was supported by AFRL and DARPA under contract FA8750-12-9-0179

language with a well-defined semantics ensures that the system design is known to meet its requirements even before it is implemented.

In previous work [11], we have successfully applied model checking to software components that have been created using model-based development (MBD) tools such as Simulink [8]. Our objective in developing the *AGREE* tool was to build on this success and extend the reach of model checking to system design models. In doing so, it is necessary to be able to deal with large, complex system models. Approaches that flatten the system model by elaborating each component and including its implementation in the same language used for the system model suffer from limited scalability. Instead, we have taken a compositional approach, attempting to exploit the verification effort and artifacts that are already part of existing software component verification processes. We do this through the use of formal assume-guarantee contracts that correspond to the requirements for each component. Each component in the system model is annotated with a contract that includes the requirements and constraints that were specified and verified as part of its development process. We then reason about the system-level behavior based on the interaction of the component contracts. The use of contracts can also be extended to architectural design patterns that have been formally verified [3]. This approach allows us to leverage our existing MBD process for software components and provides a scalable way to reason about the system as a whole.

Much work on the development of AGREE was conducted as part of the High-Assurance Cyber Military Systems (HACMS) project [2]. One of the main goals of HACMS was to show that formal methods are both practical and effective for producing safe and secure embedded software. Part of accomplishing this goal was to demonstrate the use of HACMS technologies on a real military aircraft. We applied our technologies to an unmanned helicopter produced by one of our HACMS collaborators. This vehicle can be used for a variety of missions, including remote surveillance, logistical resupply, and medical evacuation.

We modeled the software architecture of the helicopter in the Architecture Analysis and Design Language (AADL), formally verified key aspects of the design, and generated flight software from the AADL model. Formal methods were used to verify the software architecture, specify and verify software components, and prove the complete functional correctness of the seL4 secure kernel used in the mission computer. The resulting software was flown in the helicopter and was able to successfully withstand cyber-attacks conducted in-flight. Further information about the vehicle, HACMS technologies, and cyber-security demonstrations can be found in [1].

Section 2 of this paper provides background on AADL. Section 3 provides more details on the AGREE language and verification tool. Section 4 describes how we used AGREE to verify part of the command authorization logic in the unmanned helicopter, which is critical for both its safety and security.

2 Background

The Architecture Analysis and Design Language (AADL) has been developed to capture the important design concepts in real-time distributed embedded systems [5]. AADL is therefore well-suited for modeling avionics systems architectures, and provides an excellent mechanism for capturing the important details of the system design. The AADL language can capture both the hardware and software architecture in a hierarchical format. It provides hardware component models including processors, buses, memories, and I/O devices, and software component models including threads, processes, and subprograms. Interfaces for these components and data flows between components can also be defined. The language offers a high degree of flexibility in terms of architecture and component detail. This supports incremental development where the architecture is refined to increasing levels of detail and where components can be refined with additional details over time. AADL models can be built and managed using the Eclipse-based Open Source AADL Development Environment (OSATE) [12]. OSATE includes a number of tools for checking model characteristics, such as schedulability and data flows.

In AADL, the architectural model includes component interfaces, connections, and execution characteristics, but not their implementation. It describes the interactions between components and their arrangement in the system, but the components themselves are "black boxes." The component implementations are described separately using model-based specification languages or traditional programming languages, which are included by reference in the architecture model. This separation of implementation and architecture is an important factor in achieving scalability for the analysis tools that we have developed.

One of our core innovations is to structure verification arguments by following the AADL descriptions of the system. We do this through the use of formal *assume-guarantee* contracts that correspond to the behavioral requirements for each subsystem or component. Assume-guarantee contracts [9] provide an appropriate mechanism for capturing the information needed from other modeling domains to reason about system-level properties. Each component in the system model is annotated with a contract that includes the requirements and constraints that are specified and verified as part of its development process. In this formulation, guarantees correspond to the component requirements. These guarantees are verified separately as part of the component development process, either by formal or traditional means. Assumptions correspond to the environmental constraints that were used in verifying the component requirements. For formally verified components, they are the assertions or invariants on the component inputs that were used in the proof process. We then reason about the system-level behavior based on the interaction of the component contracts. Contracts provide a layer of abstraction, allowing architectural-level reasoning to be done in a top-down fashion: before a component is even implemented, its contract can be used in reasoning about how the component interacts with its environment.

3 Overview of AGREE

AGREE is a language and a tool for compositional verification of AADL models. The behavior of a model is described by *contracts* specified for each component. A contract contains a set of *assumptions* about the component's inputs and a set of *guarantees* about the component's outputs. The guarantees of a component must be true provided the component's assumptions are true. The goal of the analysis is to prove that a component's contract is entailed by the contracts of its subcomponents. Contracts of a leaf-level component must be verified to hold by its implementation.

AADL contains special syntax elements called *annexes* that can be used to extend the language. The syntax for a component's contract exists in an AGREE annex placed inside of the component type. AGREE syntax can also be placed inside of annexes in a component implementation or an AADL Package. Syntax placed in an annex in an AADL Package can be used to create libraries that can be referenced by other components.

AGREE is implemented as an Eclipse plug-in and is distributed with the OSATE AADL modeling environment. AGREE makes use of either JKind or Kind 2 for model checking, and can be configured to use several different SMT solvers. Source code for AGREE can be downloaded from http://github.com/smaccm. A complete description of the AGREE grammar can be found at [6]. Here, we identify selected AGREE statements that are used in this work.

- **Guarantee Statement**: Guarantee statements are proven by the guarantees present in subcomponent contracts. They in turn are used to prove the guarantees of a components one step above them in the model hierarchy.
- **Assume Statement**: Assume statements are used to prove the guarantees of the contract as well as the assumptions of the subcomponent contracts.
- **Initially Statement**: Initially statements are used to constrain the values of the component outputs and intermediate variables before the components clock ever ticks. This is very subtle and only matters in models that are not synchronous. The purpose of these statements will become clear when we explain how we modeled the unmanned helicopter requirements in Section 4.
- **Assert Statement**: Assert statements make unchecked statements about how the component behaves. For the purpose of analysis, assertions are treated just like system assumptions. However, unlike subcomponent assumptions, AGREE never verifies that assertions actually hold. Assert statements can also be used to refer to subcomponent variables in contracts higher up in the model hierarchy. We show examples of this in Section 4.
- **Lemma Statement**: Lemma statements are proven just like guarantees. These are used to help the model checker learn facts to improve its ability to prove other properties. They differ from guarantees in that subcomponent lemmas are not used to prove other subcomponent guarantees or system guarantees.

AGREE was originally developed to reason about systems that execute synchronously. These systems have straightforward translations to *Lustre*, a syn-

chronous dataflow language interpreted by the model checkers used by AGREE. However, many systems that are modeled in AADL do not behave synchronously. Ideally one can implement a communication protocol between components, such as Physically Asynchronous Logically Synchronous (PALS) [10], that allows the abstraction of synchronous communication to be sound. However, for many systems this is not the case.

The value of a component's clock affects its state in the following way:

- When a component's clock transitions from false to true (or is set to true in the initial state) the component's inputs are "latched". That is, from the component's perspective its inputs do not change until the next time its clock transitions from false to true.
- When a component's clock transitions from true to false, its state may change. The next state depends on the values of its current state, its latched input values, and the constraints given by its guarantees (provided that its assumptions have been historically satisfied).

In order to model non-synchronous systems, we have added additional features to the tool to allow users to place components on different clock domains. Users can then specify constraints to dictate when a component's clock may tick. Specifically, the following types of statements have been added to the language:

- **Synchrony Statement**: Synchrony statements describe the order in which the subcomponents execute. The synchrony statement expects an integer value, which indicates the number of times a subcomponent's clock can tick since any other clock has ticked. In place of a single integer, a minimum and maximum value can be provided to specify the possible range of ticks that can occur. The keywords `simult` or `no_simult` can optionally be placed at the end of the statement to indicate that any two subcomponent clocks respectively must or must not tick simultaneously. The latter would be used when specifying multiple threads that are scheduled to run on a single processor, for example. The `asynchronous` statement indicates that a component's clock is unconstrained with respect to the clocks of other components. The `latched` statement indicates that components record their inputs on the rising edge of the clock and then transition their state on the falling edge. The latched synchrony mode is useful for modeling systems where components do not have a globally consistent view of their inputs.
- **Calendar Statement**: Calendar statements comprise a comma-delimited list of subcomponents, arranged by order of their execution.

4 Application: Unmanned Helicopter

The mission computer software architecture of the unmanned helicopter application is illustrated in Figure 1. Some of the important security properties are related to proper handling of requests from ground control stations. The ground control station uses the Standardization Agreement (STANAG) 4586 protocol

for controlling both the vehicle and its surveillance camera payload. The mission computer software includes two Vehicle Specific Modules (VSM) that handle requests from the ground. The Flight VSM is responsible for control of the aircraft and the Camera VSM is responsible for operation and positioning of the surveillance camera. The ground station is referred to as the Common Unmanned Control Station (CUCS).

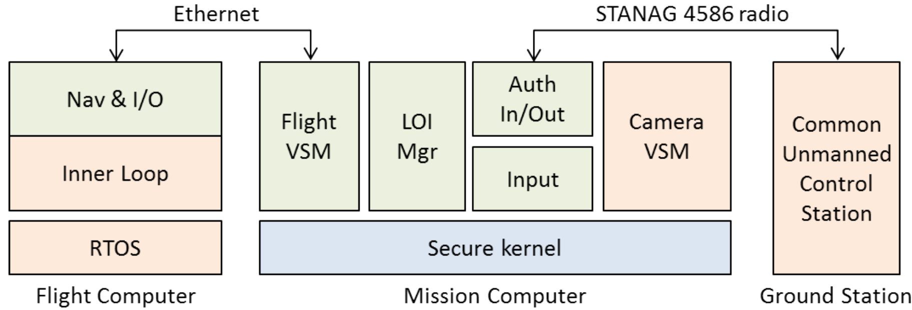

Fig. 1. Software Architecture of the Unmanned Helicopter

For this application, we will only discuss the behavior of the Encrypt/Decrypt authentication components (`authin` and `authout`), the Level of Interoperability (LOI) component (`loi`), the Input component (`input`), and the Flight Control Computer (FCC) component (`fcc`). These five components are responsible for the following tasks:

1. `authin`: The authentication in component receives incoming STANAG 4586 messages. If the message is determined to be “valid” (it decrypts and passes authentication) then the message is forwarded to the `loi` component.
2. `authout`: The authentication out component receives STANAG 4586 messages from `loi` component and forwards them to a ground station.
3. `loi`: The LOI component receives STANAG 4586 messages from various components and forwards them to other components based on rules defined in the STANAG 4586 protocol.
4. `input`: The Input component receives STANAG 4586 messages, parses them, and then sends relevant information to the `fcc` component.
5. `fcc`: The FCC component is responsible for sending information to the FCC, and relaying status information from the FCC to other components.

4.1 Modeling Assumptions

AADL is a very complex language, and defining a formal semantics for all of its constructs is exceedingly difficult. AGREE tries to strike a balance by using the

scaffolding provided by AADL to constrain the communication paths between components. AGREE requires the user to specify more complex notions about a model that are hard to generally infer from AADL models. For example, AGREE assumes by default that connections between components indicate equality between the variables on the source and destination of the connection. However, AGREE does not automatically generate constraints about how threads in a system are scheduled. It is up to the user to explicitly list these constraints in the form of assertions in a component's implementation.

AGREE is also limited by the specification language it generates (Lustre) and the model checkers that it uses to prove properties. To tractably model the behavior of the helicopter software, we made several assumptions about software generated from an AADL model. These assumptions are captured by assertions that we introduced to model the execution of the software's threads:

1. Threads do not produce new outputs until they have completed executing.
2. Threads do not preempt each other.
3. Serialization and deserialization of messages between components is implemented correctly.
4. Data sent between components is not queued. If a new message is received, it overwrites the previous message.

The first assumption is acceptable for this model because the components that we are modeling only produce a single output event per dispatch. We do not need to consider executions where a component's outputs are produced at different times during execution because a component will only ever produce a single output during execution.

The second assumption is actually false since all of the components in this model run at the same priority, and the OS will switch between them. However, this assumption is sound with respect to the properties that we attempt to prove for this model. Specifically, we prove bounds on the amount of time it takes for events to propagate through the system. Because the execution time of each component in the model is orders of magnitude smaller than its period, we do not believe that the model is over-constrained enough to eliminate concrete counterexamples to these properties. If we increase the worst case execution time of each component in our model to be the sum of the worst case execution time of all threads and we are still able to prove the properties of interest, then the properties should hold for the actual software.

The third assumption allows us to more easily reason about the types of data that are transmitted between components. Because the mission software flattens STANAG 4586 messages into arrays before transmitting them between components, it is difficult for AGREE to reason about the structure of the data. Instead we place assertions in the main component implementation to constrain the source and destination values of these messages to be the same. These assertions are shown in Figure 2.

The final assumption is due to a current limitation of AGREE. A separate timing analysis is necessary to ensure that data is not queued in the real system.

```
--constrain the interfaces between components
assert loi.auth_in = authin.auth_in;
assert loi.mid = authin.stanag_mid;
assert input.stanag_mid = loi.mid;
assert loi.to_auth_stanag_mid = authout.stanag_mid;
assert input.sender_mid = loi.from_sender_mid;
```

Fig. 2. Constraints to force variables representing fields of incoming and outgoing data the same between components

4.2 Scheduling Constraints

Typically, when a binary is generated from an AADL model, certain annotations in the form of AADL properties are used to determine how to schedule the threads present in the model. However, AGREE does not use these annotations to automatically generate constraints for the clocks associated with each thread. Instead the user must manually assert constraints about how the system executes. There are two reasons why AGREE does not automatically generate these constraints:

– The exact semantics of the system may be difficult or impossible to model accurately in AGREE. This is primarily true for components that have more than one dispatch. AGREE implicitly assumes that each AADL component has a single thread of execution.
– A user may wish to express a set of constraints that is more *abstract* than the true scheduling semantics. This can make it easier to prove properties that are true for both the abstraction and the concrete executions of the model.

The constraints shown in Figures 3 and 4 were used to model the scheduling semantics of the components running on the mission computer. In order to simplify the specification we introduced Boolean variables with suffix `_clk_rise` and `_clk_fall` to represent when the clock of a component has a rising or falling edge.

The node `rise` defined in the `AgreeTypes` package evaluates to true if and only if its input was false on the previous step and true on the current step. Similarly the node `fall` defined in the `AgreeTypes` package evaluates to true if and only if its input was true on the previous step and false on the current step. On the initial state `rise` evaluates to true if its input is true and `fall` evaluates to true if its input is false.

Figure 3 shows the constraints that we used to dictate when a component may begin executing (when its clock rises). Threads modeled in AADL may dispatch under two conditions: 1) the thread receives an event on an input event (or event data) port or 2) the thread is dispatched by a periodic timer. The first assertion in Figure 3 constrains the `loi` thread to begin executing if and only if the `authin` component sends a STANAG 4586 message to the `loi` component or the `input` component sends a STANAG 4586 message to the `loi` component.

The second assertion constrains the `authout` component to only begin executing when it receives a STANAG 4586 message from the `loi` component. The third assertion forces the `authin` component to run periodically with a constant time bound. This bound is the assumed frequency of incoming messages. The final two assertions constrain the `input` and `fcc` components to run periodically at a rate of 100ms.

```
--    non-periodic components
assert loi_clk_rise =
  ((event(authin.stanagout) and authin_clk_fall) or
    (event(input.sender) and input_clk_fall)
  );
assert authout_clk_rise = (event(loi.loi2auth) and loi_clk_fall);

--    periodic components
assert condition authin_clk_rise occurs each VSMPkg.commsec_bound;
assert condition input_clk_rise occurs each 100000.0;
assert condition fcc_clk_rise occurs each 100000.0;
```

Fig. 3. Constraints about when a component may begin executing

The constraints in Figure 3 assert that the components must run when certain events occur (either a periodic dispatch occurs or an event arrives on an input). To model the components' minimum and maximum execution times we make the additional assertions shown in Figure 4.

```
assert whenever input_clk_rise occurs input_clk_fall occurs during [10.0, 50.0];
assert whenever fcc_clk_rise occurs fcc_clk_fall occurs during [10.0, 50.0];
assert whenever authin_clk_rise occurs authin_clk_fall occurs during [10.0, 50.0];
assert whenever authout_clk_rise occurs authout_clk_fall occurs during [10.0, 50.0];
assert whenever loi_clk_rise occurs loi_clk_fall occurs during [10.0, 50.0];

assert whenever authout_clk_rise occurs authout._CLK holds during [0.0, 10.0);
assert whenever loi_clk_rise occurs loi._CLK holds during [0.0, 10.0);
```

Fig. 4. Constraints about when a component may stop executing

The first five assertions guarantee that a component will complete execution during the interval specified by its minimum and maximum execution times. However, for the `authout` and `loi` components we must also assert that their clocks remain high until at least their minimum execution time. This is because the semantics of the pattern used in this assertion does not constrain the second event to *only* occur during the specified interval. However, for the `input`, `fcc`, and `authin` components the periodic constraints listed in Figure 3 implicitly prevent the clock from rising again before its minimum execution time.

4.3 Component Contracts

Next we describe the assume-guarantee contracts for several critical components.

The LOI component. The AGREE annex in the LOI component houses the most detailed contract. The assumptions and guarantees that are present in the process were derived primarily from the STANAG 4586 specification. The LOI component is responsible for keeping track of the current LOI and the ground station that is in control of the vehicle. It is also responsible for forwarding STANAG 4586 messages to the correct components in the system.

In total, the LOI component makes three guarantees:

1. If no message is received, or the message that is received is not an authorization request, then all of the LOI state variables remain the same. This property is likely implicit to any software implementation of the component, but we must make it explicit or else the model checker will choose non-deterministic values for these variables.
2. If a message is received and it is an authorization request, then it is handled according to the STANAG 4586 specification. Specifically, this guarantee covers the following scenarios:
   (a) If the CUCS who is in control is relinquishing control, then no one is overriding control, no one is in control, and the LOI is set to zero.
   (b) If a CUCS is requesting control or attempting to override control and the previous LOI is 3 and the requested LOI is greater than 3, then the CUCS is granted control.
   (c) If a CUCS is requesting control and no CUCS is currently overriding control, then the CUCS is granted control.
   (d) If a CUCS is attempting to override control and no CUCS is currently overriding control, then the CUCS overrides control.
3. If the message is received and the current LOI is approved for the message type, then it is forwarded to the appropriate VSM. If the LOI is 3 and the control station is set to the camera VSM, then the message is forwarded to the camera VSM. Otherwise, the message is forwarded to the flight VSM. If no message is received or if the message is not approved at the current LOI then the message is not forwarded to either VSM.

These guarantees could possibly be broken out into smaller requirements rather than large nested "if-then-else" blocks. This is more of a choice of style and readability. The LOI component assumes that the data fields for authorization messages are in their correct ranges. This assumption should be satisfied by guarantees from the `authin` component.

The Authentication In component. The contract for the `authin` component is shown in Figure 5. The component guarantees that it only produces a STANAG 4586 messages on its output if it received a commsec message on its

input. This restricts the component to only output STANAG 4586 messages if it just received a commsec message. The component also guarantees that any authorization message that it passes on to the LOI component has valid data. By valid data we mean that specific fields in an authorization message are within the ranges specified by the STANAG 4586 specification. This is needed to prove the assumption listed in the `loi` component. We use an `initially` statement to say that before the component's clock ticks it has no events being sent on its STANAG 4586 output.

We represented different STANAG 4586 message types by including multiple subcomponents within the STANAG 4586 message data implementation. Implementing the message data this way is similar to how someone would implement it as a structure in the C language using a union for different structures over the message data field. In the contract for the `authin` component we use the variable `auth_in` to specifically reference this portion of the STANAG 4586 message data field.

```
eq auth_in : AgreeTypes::STANAG_4586_message.cucs_auth_req;
eq stanag_mid : int;

initially:
  not event(stanagout);

guarantee "we only send a message out if we get one in" :
  event(stanagout) => event(commsecin);

guarantee "valid auth data" :
  (0 < auth_in.rloi and auth_in.rloi <= 5 and
   0 <= auth_in.csm and auth_in.csm <= 2 and
   0 < auth_in.cucsid and auth_in.cucsid < 255 and
   --right now we model just two control stations
   0 <= auth_in.cs and auth_in.cs <= 1);
```

Fig. 5. The contract of the `authin` component

The Input component. The `input` component is responsible for decoding STANAG 4586 messages and forwarding commands to the FCC component. It also determines which "mode" the vehicle is in. The `input` component transitions to various modes based on the different STANAG 4586 messages it receives from the `loi` component and information it receives about the state of the vehicle from the FCC. The `input` component also reports some status messages back to the ground station via the `loi` component.

The contract of the `input` component is shown in Figure 6. We have introduced AGREE variables in the `input` component's contract to model the message ID of incoming STANAG 4586 messages, the vehicles mode, a status flag indicating whether or not a waypoint was sent to the `fcc` component, the

```
--TODO fill in the logic of this component
agree_input stanag_mid : int;
eq mode : int;
eq waypoint_sent_to_fcc : bool;
eq sender_mid : int;
eq route_accepted : bool;

initially:
  mode = VSMPkg.NO_MODE and
  not waypoint_sent_to_fcc;

  guarantee "initially the vehicle starts in NO_MODE":
    (mode = VSMPkg.NO_MODE) -> true;

  guarantee "the vehicle never transtions back to NO_MODE":
    true -> not pre(mode = VSMPkg.NO_MODE) => not (mode = VSMPkg.NO_MODE);

  --this guarantee just abstracts the meaning of a waypoint being sent to the fcc
  guarantee "a waypoint message is only sent to the fcc if something is sent to the fcc" :
    waypoint_sent_to_fcc => event(send2fcc);

  guarantee "whenever a waypoint is sent to the fcc an acknowledgement is sent to the loi" :
    waypoint_sent_to_fcc => sender_mid = 900 and event(sender);

  guarantee "a received route is always accepted" :
    route_accepted = (event(loi2vehicle) and stanag_mid = 801);

  guarantee "if we transition to MANUAL WAYPOINT MODE it is because we saw certain message ids" :
    true -> (mode != pre(mode) =>
      (event(loi2vehicle) and stanag_mid = 42) or
       mode = VSMPkg.WAYPOINT_MODE and pre(mode) = VSMPkg.LAUNCH_MODE);
```

Fig. 6. The contract for the `input` component

message id of outgoing STANAG 4586 messages, and a status flag indicating whether or not an uploaded route is accepted.

The guarantees of the component contract describe the state transitions that the component makes as well as when information is forwarded to the `fcc` and `loi` components.

The FCC and Authentication Out components. The contracts for the `fcc` and `authout` components do not contain any assumptions nor any guarantees. However, we have defined several AGREE variables in the contracts to represent state variables of the components. This allows us to specify guarantees in the top level contract about when data is sent from and arrives at these components.

4.4 Properties

Provable Guarantees. The first property that we prove about the system is that "whenever an authorization message is received and the current LOI is 3 the vehicle accepts the message within a specified latency." The formalization of this property is shown in Figure 7. An authorization message is STANAG 4586 message with a message ID of 1. We define the time in which an authorization message is received as the time that the `authin` finishes executing and forwards a STANAG 4586 message to the LOI component with a message ID of 1. The current LOI is the value of the current LOI in the LOI component at the time

the authorization message is received. We consider an authorization message to be accepted if the LOI component changes the current LOI and the ID of the CUCS in control to be the values requested in the authorization message. In order to prove this property we set the specified latency to 200ms.

```
guarantee "loi greater than three always gets control" :
  whenever
    received_auth_message_3
  occurs
    acted_on_auth_message
  occurs during [0.0, VSMPkg.system_latency];
```

Fig. 7. The guarantee that LOI greater than 3 always gets control.

```
guarantee "Do not accept NAV commands with loi less than 4":
  vehicle_received_stanag and
  vehicle_stanag_mid >= 800 and
  vehicle_stanag_mid < 1000 =>
  current_loi >= 4;
```

Fig. 8. The guarantee that NAV commands with loi less than 4 are not accepted.

The second property that we prove is shown in Figure 8. This guarantee states that if a navigation command reaches the `input` component then the current LOI is 4. The LOI component guarantees that a navigation message is only forwarded to the `input` component if the LOI is 4.

Figure 9 shows properties that depend only on the state machine described by the guarantees of the `input` component. The state of the mode variable in the `input` component depends on the previous mode and any STANAG 4586 messages that are received. Currently we do not have a complete description of the state machine so we are only able to prove two of the properties. To prove the latter two properties we would need to strengthen the contract of the `input` component to describe in more detail how state transitions occur.

Possible Counterexamples. There were several properties that we assumed were true about this model, but for which the tool was able to produce counterexamples. The first of these properties is shown in Figure 10. The guarantee states that the vehicle cannot transition into `MANUAL_WAYPOINT_MODE` unless the current LOI is 4 or 5. Intuitively this should be true because in order to transition into `MANUAL_WAYPOINT_MODE` the LOI component must forward a STANAG 4586 message that requires an LOI of at least 4.

However, the tool produces a counterexample for the following scenario:

```
guarantee "The aircraft is initially in NO_MODE" :
  (mode = VSMPkg.NO_MODE) -> true;

guarantee "The aircraft never transitions back into NO_MODE" :
  true -> mode != pre(mode) => mode != VSMPkg.NO_MODE;

guarantee "The aircraft can only enter SLAVE2SENSOR mode from WAYPOINT or LOITER mode" :
  true ->
    (mode = VSMPkg.SLAVE2SENSOR_MODE and not pre(mode = VSMPkg.SLAVE2SENSOR_MODE) =>
      pre(mode = VSMPkg.WAYPOINT_MODE) or
      pre(mode = VSMPkg.LOITER_MODE));

guarantee "The aircraft can only enter MANUAL_WAYPOINT mode from WAYPOINT or LOITER mode":
  true ->
    (mode = VSMPkg.MANUAL_WAYPOINT_MODE and not pre(mode = VSMPkg.MANUAL_WAYPOINT_MODE) =>
      pre(mode = VSMPkg.WAYPOINT_MODE) or
      pre(mode = VSMPkg.LOITER_MODE));
```

Fig. 9. Guarantees about the state machine in the `input` component.

```
guarantee "The aircraft requires LOI of 4 or 5 in order to transition into MANUAL_WAYPOINT_MODE":
  true -> mode != pre(mode) and mode = VSMPkg.MANUAL_WAYPOINT_MODE =>
    current_loi = 5 or
    current_loi = 4;
```

Fig. 10. A guarantee about mode transitions under a certain LOI

1. The current LOI is 4 and the `LOI` component receives a STANAG 4586 message with ID #42. Because the current LOI is 4 this message is forwarded to the `input` component.
2. The `input` component receives this STANAG 4586 message with ID #42 message and begins transitioning its mode.
3. Before the `input` component finishes the `loi` component receives a request to relinquish control (setting LOI to 0).
4. The `input` component completes execution and the vehicle is now transitions to `MANUAL_WAYPOINT_MODE` with LOI 0.

While this counterexample does not seem spurious, it might instead illustrate an error in our formalization of the property. We probably do not care about the value of the LOI the instant that the mode transition occurs. Instead we care that the mode transition occurs in response to a STANAG 4586 message that was received while the LOI was 4 or 5.

The other property that produces a counterexample is shown in Figure 11. The tool produces a counterexample where the following scenario occurs:

1. The `authin` component receives a STANAG 4586 message and forwards it to the `loi` component.
2. The `loi` component has LOI of 4 and forwards the message to the `input` component.
3. The `loi` component forwards a message to the `authout` component. This “erases” the event signal from the `loi` component to the `input` component.
4. The `input` component executes without receiving the route message.

The reason that the tool produces this counterexample is because we do not accurately model the queuing behavior of the real software. In our AGREE model previous messages are overwritten by new messages.

```
eq route_message_received : bool =
  stanag_message_received and
  (current_loi = 4 or current_loi = 5) and
  (incoming_stanag_mid = 801);

guarantee "A route can be uploaded to the aircraft regardless of state (but correct LOI)":
  whenever
    route_message_received
  occurs
    input_accepts_route
  occurs during [0.0, VSMPkg.system_latency];
```

Fig. 11. A guarantee about routes being uploaded to the aircraft

5 Conclusion and Future Work

In this paper we have described the AGREE language and tool for compositional analysis of systems modeled in AADL. AGREE translates a system architecture model annotated with assume-guarantee contracts into a collection of model checking problems, proceeding through all the layers of a hierarchical system model. The compositional "divide and conquer" approach allows us to analyze large and complex avionics systems. Embedding our approach in AADL allows us to integrate with existing MBD tools for software component development [7], and permits system software to be generated directly from the models that have been verified.

To demonstrate the practicality and effectiveness of our approach, we have demonstrated its use on an unmanned military helicopter. We modeled the software architecture of the helicopter in AADL and used AGREE to verify part of the command authorization logic used for communicating with the ground station. Flight software for the helicopter was generated from the verfied AADL model and was flown in the helicopter, successfully withstanding cyber-attacks in-flight.

We are continuing development of AGREE as part of DARPA's Cyber Assured Systems Engineering program (CASE). The goal of CASE is to develop the necessary design, analysis and verification tools to allow system engineers to design-in cyber resiliency and manage tradeoffs as they do other nonfunctional properties when designing complex embedded computing systems. We are using AGREE to verify architectural patterns and components for that address cybersecurity requirements. We are integrating AGREE with assurance cases that are captured using the Resolute tool for embedding assurance cases in AADL models [4]. We are also implementing a number of language extensions to AGREE, including support for array data and information flow.

References

1. D. Cofer, J. Backes, A. Gacek, D. DaCosta, M. Whalen, I. Kuz, G. Klein, G. Heiser, L. Pike, A. Foltzer, M. Podhradsky, D. Stuart, J. Graham, and B. Wilson. Secure Mathematically-Assured Composition of Control Models. Technical Report HACMS Final Report, AFRL-RI-RS-TR-2017-176, October 2017.
2. D. Cofer, A. Gacek, J. Backes, M. W. Whalen, L. Pike, A. Foltzer, M. Podhradsky, G. Klein, I. Kuz, J. Andronick, G. Heiser, and D. Stuart. A formal approach to constructing secure air vehicle software. *IEEE Computer Magazine (to appear)*, Jan. 2019.
3. D. D. Cofer, A. Gacek, S. P. Miller, M. W. Whalen, B. LaValley, and L. Sha. Compositional Verification of Architectural Models. In *NFM 2012*, volume 7226, pages 126–140, April 2012.
4. A. G. et. al. Resolute: An assurance case language for architecture models. In *HILT 2014*, pages 19–28, New York, NY, USA, 2014. ACM.
5. P. Feiler and D. Gluch. *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*. Addison-Wesley Professional, 2012.
6. A. Gacek, J. Backes, M. Whalen, and D. Cofer. AGREE users guide. https://github.com/smaccm/smaccm, 2018.
7. J. Liu, J. D. Backes, D. D. Cofer, and A. Gacek. From design contracts to component requirements verification. In *NASA Formal Methods - 8th International Symposium, NFM 2016, Minneapolis, MN, USA, June 7-9, 2016, Proceedings*, pages 373–387, 2016.
8. MathWorks. The MathWorks Inc. Simulink Product Web Site. http://www.mathworks.com/products/simulink, 2004.
9. K. L. McMillan. Circular compositional reasoning about liveness. In *Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, CHARME '99, pages 342–345, London, UK, UK, 1999. Springer-Verlag.
10. J. Meseguer and P. C. Ölveczky. Formalization and correctness of the pals architectural pattern for distributed real-time systems. *Theor. Comput. Sci.*, 451:1–37, Sept. 2012.
11. S. P. Miller, M. W. Whalen, and D. D. Cofer. Software model checking takes off. *Commun. ACM*, 53(2):58–64, Feb. 2010.
12. The Software Engineering Institute. OSATE: Plug-ins for front-end processing of AADL models, 2013.