Using AGREE to model LOI logic in the ULB

John Backes

Rockwell Collins, Bloomington MN 55438

Abstract. The purpose of this document is to discuss how AGREE has been used to model the LOI authorization portion of STANAG 4586 running on the Unmanned Little Bird (ULB). We expect that the reader has some familiarity with AADL and preferably some familiarity with the synchronous dataflow programming language Lustre. We give a very brief overview of STANAG 4586. Specifically we explain the goals of the LOI validation portion of the CUCS authorization messages. We then describe the relevant portions of the ULB architecture that reason about this portion of the protocol. Finally, we give a description of how AGREE was used to prove properties about how the vehicle's subcomponents interact in response to authorization requests via the STANAG 4586.

1 Overview of STANAG 4586

The Unmanned Little Bird (ULB) communicates with ground stations (referred to as Core UAV Control Systems (CUCS)) via STANAG 4586. STANAG 4586 is designed to support shared communication and control of various UAVs with differing capabilities operating in a theatre with multiple forces. The STANAG 4586 model UAV architecture consists of a single Vehicle Specific Module (VSM) which communicates with a CUCS. The role of a UAV's VSM is to translate STANAG 4586 messages into messages that the UAV hardware can interpret and act upon. Likewise, the VSM sends data via STANAG 4586 back to a CUCS.

STANAG 4586 consists of messages that demand differing levels of control of the vehicle. In order to allow multiple CUCS to interact with a UAV, messages require different levels of interoperability (LOI) to interact with the vehicle. The allowed LOI for each CUCS is decided by the vehicle at the beginning of a mission. A CUCS can then request a different LOI from a UAV during its mission, and the UAV will respond accordingly.

It is crucial that the vehicle not act upon messages sent by a CUCS with an inadequate LOI. Likewise it is important that a UAV only grant an LOI to a CUCS that is appropriate based on the current state of the vehicle and the permissions decided upon at the beginning of the mission. We have used our AGREE tool to model how the state of ULB changes in response to STANAG 4586 messages and to prove that these properties hold.

2 Architecture

The ULB architecture differs slightly from the examples given in the STANAG 4586 specification; the ULB architecture has two VSMs: one for communicating

with the Flight Control Computer (FCC) and one for communicating with the Wescam. However, the same communication channel is used to communicate with both VSMs.

The Phase II architecture (Shown in Figure 1) contains four components running on seL4: the FCC VSM component, the LOI Manager component, the Authentication Component, and the Wescam VSM component. In the AADL model, we modeled these components as processes (though there has been some discussion about modeling these components differently in AADL). The subcomponents within the two VSMs are modeled as threads. Most of the other components shown in Figure 1 are present in the AADL model, but are not reasoned about with AGREE.

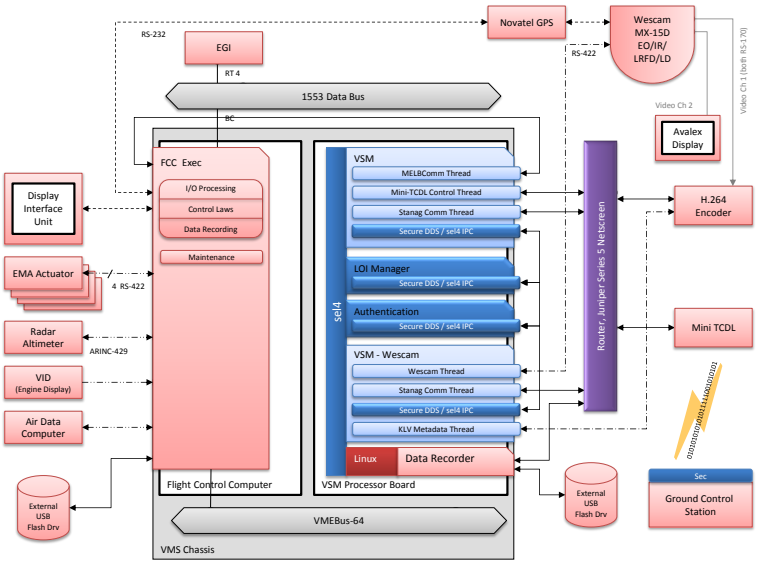

Fig. 1. The Phase II ULB Architecture *Thanks to Boeing for the Figure

Figure 1 is not completely isomorphic to the AADL architecture. Other design decisions have been made since the figure was created that affect the communication mechanisms between different software components. For example, the LOI Manager directly processes STANAG messages and then passes them on to other components. The Authentication Component is responsible for decrypting/encrypting messages received and sent from the LOI manager.

We had to modify Boeing's AADL architecture slightly in order to reason about it with AGREE. Specifically, we currently do not have mechanisms in AGREE to reason about bidirectional connections. We replaced bidirectional

connections with two corresponding unidirectional connections. This restriction is something we plan to address in the future.

3 Overview of AGREE

3.1 Compositional Verification

AGREE is a language and a tool for compositional verification of AADL models. The behavior of a model is described by *contracts* specified on 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 are 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. Formally, let a system $S : (A, G, C)$ consist of a set of assumptions A, guarantees G, and subcomponents C. We use the notation S_g represent the conjunction of all guarantees of S (i.e., $S_g = \bigwedge_{g \in G} g$) and S_a represent the conjunction of all assumptions of S (i.e., $S_a = \bigwedge_{a \in A} a$). Each subcomponent $c \in C$ is itself a system with assumptions, guarantees, and subcomponents. The goal of our analysis is to prove that a system's guarantees hold as long as its assumptions have always held. This is stated formally in Formula 1.

$$H(S_a) \rightarrow S_g \tag{1}$$

Where the predicate H is true if its argument has held *historically* (i.e., the expression has been true at every time step up until and including now). In order to prove Formula 1 we prove that the assumptions of all the subcomponents of system S hold under the assumptions of S.

$$H(S_a) \rightarrow c_a \tag{2}$$

Formula 2 should be shown for each subcomponent $c \in C$. This formula is actually stronger than what we need to prove. It might be the case that the assumptions of certain subcomponents are satisfied by the guarantees of other subcomponents (and possibly the guarantees of the component itself at previous instances in time). This weaker condition is shown in Formula 3.

$$H(S_a) \wedge Pre(H(c_g)) \wedge \bigwedge_{d \in C, c \neq d} H(d_g) \rightarrow c_a \tag{3}$$

Formula 3 should be shown for each subcomponent $c \in C$. This formula is also not quite what we want to prove. It is not sound in certain scenarios where components are connected immediately in a cyclic manner. One could imagine a scenario where the assumptions of of two components are true precisely because of the guarantees of the other component (i.e., $c_g \rightarrow d_a$ and $d_g \rightarrow c_a$ for $c, d \in C$ and $c \neq d$). Perhaps components c and d assume that their inputs are positive, and they guarantee that their outputs are positive. If their outputs are wired to the other's inputs, the state of the system is poorly defined. To avoid this problem, AGREE actually creates a total ordering of a system's subcomponents. It uses this ordering to determine which subcomponent guarantees are used to prove the assumptions of other subcomponents. This slight modification of Formula 3 is shown in Formula 4.

$$H(S_a) \wedge Pre(H(c_g)) \wedge \bigwedge_{d \in C, d < c} H(d_g) \rightarrow c_a \tag{4}$$

In practice this order is determined by the order in which the subcomponents are listed in an AADL component implementation. Subcomponents listed later are higher in the order than those listed sooner. If Formula 4 holds for all subcomponent assumptions then we demonstrate that Formula 1 is true by showing that the system assumptions and subcomponent guarantees satisfy the system guarantees. Formally, if Formula 4 holds for all $c \in C$, then Formula 5 implies Formula 1.

$$H(S_a) \wedge \bigwedge_{c \in C} H(c_g) \rightarrow S_g \tag{5}$$

The reader can review [**?**] for rigorous arguments about the correctness of the analysis. To motivate how this analysis is applied to a real-world model, we present a simplified example of a portion of the ULB's AADL model in Figure 2. At the top of the model's component hierarchy is a system component representing the ULB. The direct subcomponents of the ULB are the Wescam VSM and the LOI Manager. The Wescam VSM process contains two threads: the STANAG Thread and the Camera Thread. Each component in the model contains an AGREE contract describing the component's behavior. For example, the contract of the STANAG Thread running in the Wescam VSM guarantees that information contained in a STANAG message is translated into information that the Camera Thread can understand. This guarantee is essentially a set of constraints describing how fields of the STANAG message are mapped to fields of an internal message describing the state of the camera.

The guarantees of the STANAG and Camera threads should be sufficient to prove the guarantees of the Wescam VSM. Specifically, it should be deduced from the contracts of the two threads that "If no STANAG message is received,

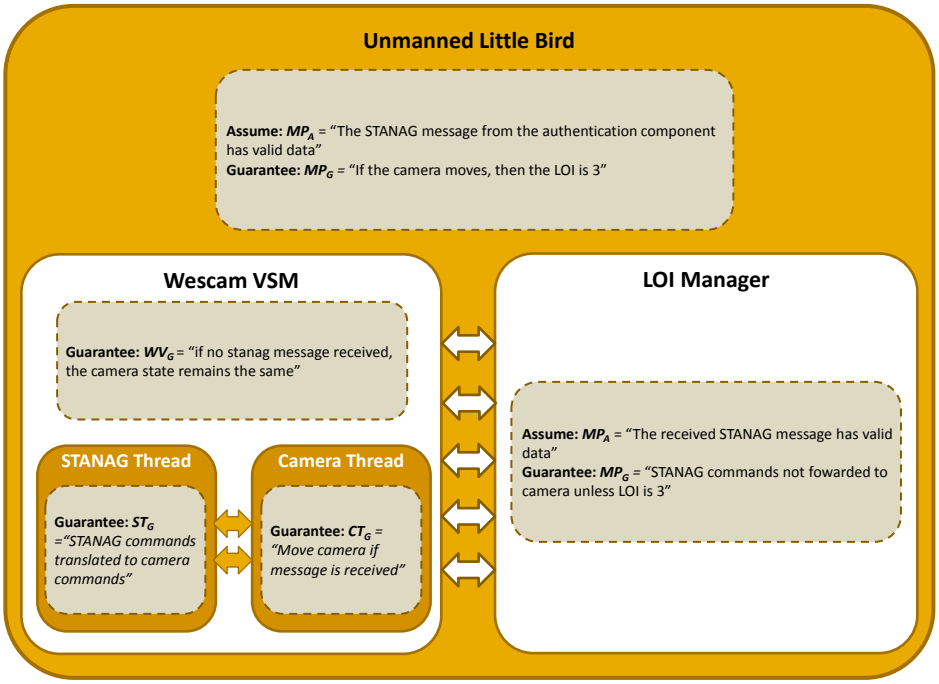

Fig. 2. A simplified picture of some of the ULB components with example AGREE contracts

then the camera state remains the same". In the next section we describe in more detail what we modelled and what we proved about the LOI authorization portion of STANAG 4586 running on the ULB.

3.2 Modeling Component Execution

By default, AGREE reasons about a model synchronously. That is, the state of a component is free to change at any instant in time. However, this constraint might be too strict to be realistic for some systems. For example, it is not accurate to model multiple threads scheduled to run on the same processor this way.

AGREE can also be used to model quasi-synchronous systems. A quasi-synchronous system is one in which components run on individual clocks and the relative rates of the clocks are bounded. For example, one might choose to model two processors that run at the same rate, but have some bounded amount of drift or jitter. We refer to the constraint on when components' clocks can tick as the "calendar". Each component has a calender that determines the order in which its subcomponents execute. We say that a components "clock ticks" to refer to the moment in time when its state is allowed to change.

Proving properties compositionally about models with components running on different clock domains is more complex than the methods presented in the previous section. Informally, the analysis is different in several ways.

1. The guarantees of a component only hold *once its clock has ticked*. A component's guarantees are always relative to its own clock. For example, if a component guarantees that "Its output is one greater than its previous input", it really means "when the components clock ticks, its output is one greater than its input immediately before its clock ticked".
2. Likewise, we only attempt to prove that a component's assumptions hold when the component's clock ticks.
3. In order for the compositional proof to be sound, a component's calendar cannot be *more restrictive* than the calendars of its subcomponents. In other words, the calendar we choose for a component should abstract the possible orders of executions of all the components lower than it in the hierarchy. This a subtle point, and we do not choose to go into too much detail about it in this document. Right now AGREE allows the user to manually choose what calendar to use at each component in the hierarchy. Realistically, the order in which the leaf level components are allowed to execute should determine the calendars of components higher up the model. Eventually we will change the tool do this automatically.

In the next section we discuss AGREE's syntax. We explain how the "synchrony statement" is used to enforce the relative rates of a components subcomponents.

3.3 The AGREE Language

AADL describes the interface of a component in a *component type*. A component type contains a list of *features*, which are the inputs and outputs of a component,

and possibly a list of AADL properties. A *component implementation* is used to describe a specific instance of a component type. A component implementation contains a list of subcomponents and list of connections that occur between its subcomponents and features.

For example, one may decide to create a component type for a car which contains features describing its throttle, speed, and direction. A car component may have many implementations (like a 2006 Toyota Camry, or a McLaren 650s). Different implementations may contain different electronic components, actuators, etc.

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.

Figure 3 shows the simplified description of AGREE's grammar in EBNF. An AGREE annex in an AADL component type contains syntax that is recognized as a `<contr>` element. Similarly, an AGREE annex in an AADL component implementation contains syntax that is recognized as a `<impl>` element. The types of statements found in an AADL component type are described below.

- **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. These are the guarantees described in Section 3.1.
- **Assume Statement**: Assume statements are used to prove the guarantees of the contract as well as the assumptions of the subcomponent contracts. These are the assumptions described in Section 3.1.
- **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. These statements will make more sense when we explain how we modelled the ULB requirements in Section 4.
- **Equation Statement**: Equation statements are used to define variables. Variables defined with equation statements are thought of as "intermediate" variables or variables that are not meant to be visible in the architectural model (unlike component outputs or inputs). Equation statements can define variables explicitly by setting the equation equal to an expression immediately after it is defined. Equation statements can also define variables implicitly by not setting them equal to anything, but constraining them with assumption, assertion, initially, or guarantee statements. Equation statements can actually define more than one variable at once by writing them in a comma delimited list. One might do this to constrain a list of variables to the results of a node statement that has multiple return values or to more cleanly list a set of implicitly defined variables.

⟨*contr*⟩ ::= ⟨*contr-stmt*⟩ ⟨*contr*⟩ | ⟨*contr-stmt*⟩

⟨*contr-stmt*⟩ ::= 'guarantee' ⟨*STR*⟩ ':' ⟨*expr*⟩ ';'
| 'assume' ⟨*STR*⟩ ':' ⟨*expr*⟩ ';'
| 'initially' ':' ⟨*expr*⟩ ';'
| ⟨*eq-stmt*⟩
| ⟨*node-stmt*⟩

⟨*impl*⟩ ::= ⟨*impl-stmt*⟩ ⟨*impl*⟩ | ⟨*impl-stmt*⟩

⟨*impl-stmt*⟩ ::= 'assert' ⟨*expr*⟩ ';'
| 'synchrony' ':' ⟨*INT-LIT*⟩ ('no_simult')? ';'
| 'lemma' ⟨*STR*⟩ ':' ⟨*expr*⟩ ';'
| ⟨*eq-stmt*⟩
| ⟨*node-stmt*⟩

⟨*eq-stmt*⟩ ::= 'eq' ⟨*arg-lst*⟩ '=' ⟨*expr*⟩ ';'
| 'eq' ⟨*arg-lst*⟩ ';'

⟨*node-stmnt*⟩ ::= 'node' ⟨*ID*⟩ '(' ⟨*arg-lst*⟩ ')'
'returns' '(' ⟨*arg-lst*⟩ ')' ';'
('var' ⟨*arg-lst*⟩)?
'let' ⟨*eq-lst*⟩ 'tel' ';'

⟨*expr*⟩ ::= ⟨*BOOL-LIT*⟩
| ⟨*INT-LIT*⟩
| ⟨*REAL-LIT*⟩
| ⟨*ID*⟩
| ⟨*ID*⟩ '(' ⟨*expr-lst*⟩ ')'
| 'pre' '(' ⟨*expr*⟩ ')'
| 'event' '(' ⟨*ID*⟩ ')'
| 'floor' '(' ⟨*expr*⟩ ')'
| 'real' '(' ⟨*expr*⟩ ')'
| 'Get_Property' '(' ⟨*expr*⟩ ',' ⟨*aadl-property*⟩ ')'
| '-' ⟨*expr*⟩
| 'not' ⟨*expr*⟩
| ⟨*expr*⟩ '{' ⟨*ID*⟩ ':=' ⟨*expr*⟩ '}'
| ⟨*expr*⟩ ('*'|'/'|'mod'|'+'|'-') ⟨*expr*⟩
| ⟨*expr*⟩ ('<'|'<='|'>'|'>='|'='|'!=') ⟨*expr*⟩
| ⟨*expr*⟩ ('and'|'or') ⟨*expr*⟩
| ⟨*expr*⟩ '=>' ⟨*expr*⟩
| ⟨*expr*⟩ '->' ⟨*expr*⟩

⟨*type*⟩ ::= 'int' | 'real' | 'bool' | ⟨*aadl-type*⟩

⟨*arg-lst*⟩ ::= ⟨*ID*⟩ ':' ⟨*type*⟩ ',' ⟨*arg-lst*⟩ | ⟨*ID*⟩ ':' ⟨*type*⟩

⟨*eq-lst*⟩ ::= ⟨*eq-stmt*⟩ ⟨*eq-lst*⟩ | ⟨*eq-stmt*⟩

⟨*expr-lst*⟩ ::= ⟨*expr*⟩ ',' ⟨*expr-lst*⟩ | ⟨*expr*⟩

Fig. 3. A simplified description of the grammar for AGREE

– **Node Statement**: Node statements are used to define "functions" that might be used frequently in a component type or implementation. Nodes cannot be recursive or mutually recursive. Nodes can have multiple return values. If this is the cause, they must be referenced by an equation statement that has multiple arguments. Nodes can also be defined in an AADL Package. If so, they can be referenced in any expression anywhere in the model. This way one can make a library of certain types of nodes that are useful for different tasks.

The `<expr>` elements in the grammar are very similar to Lustre expressions [**?**]. Expressions reason about the current state or past states of variables and can reference variables defined in equation statements or other identifiers in the AADL model. The following is a list of different types of expressions. We describe them in the order they are listed as alternatives to the grammar rule for `<expr>` in Figure 3.

– **Boolean Literal**: `true` or `false`
– **Integer Literal**: `1`, `42`, `-1337`, etc ...
– **Real Literal**: `3.1415`, `1.6180`, `0.001`, etc ...
– **ID Expression**: An ID is an alpha numeric string (not starting with a number). ID expressions can also have dots in them (e.g., `foo.bar.biz`). ID expressions correspond to variables defined in equation statements or AADL features. AGREE reasons about the AADL `Base_Types::Integer`, `Base_Types::Float`, and `Base_Types::Boolean` as `int`, `real`, and `bool` types, respectively. ID expressions with dots in them correspond to record types or variables of a subcomponent. For example, one could use the ID expression `foo.bar` to reference the input, output, or equation variable `bar` of subcomponent `foo` within the implementation of some AADL component. AGREE reasons about AADL Data Implementations like record types. This will be more clear when we discuss how we modeled the STANAG 4586 messages in Section 4.
– **Node Call Expression**: A node call expression is an ID of a defined node followed by parenthesis. If the node is defined in an AADL Package, then the ID should be the AADL Package name a dot (`.`) and then the node name.
– **Previous Value Expression**: A previous value expression evaluates to the value of its argument on the previous time frame. It should that it be guarded by an arrow expressions as its value is undefined on the initial step.
– **Event Expression**: An event expression is a special predicate that is used to reason about AADL event data ports. For an input event data port, its semantics are such that it evaluates to true if a value is *present* on the event port and false otherwise. For an output event data port, its semantics are such that it evaluates true if data is being sent on the port and false otherwise.
– **Floor Expression**: A floor expression takes an expression of type `real` as an argument and returns an `int` equal to the floor of the number.
– **Real Expression**: A real expression takes an expression of type `int` as argument and returns a `real` equal to its value.

- **Get Property Expression**: A get property expression allows a user to reason about values of AADL properties in the model. The first argument is the relative path to an AADL component in the instance model or '`this`' if the property exists in the component in which get property statement lives. The second argument is the name of the AADL property.
- **Minus Expression**: This expression is used to negate integer or real valued expressions.
- **Not Expression**: This expression is used to negate boolean valued expressions
- **Record Update Expression**: The record update expression expects an expression of record type on the left hand side of the curly braces. It returns the same record as the left hand side expression except with its member "ID" set to the value of the expression on the right hand side of the '`:=`';
- **Arithmetic Operations**: Arithmetic operations must be performed on expressions of the same type. They follow the standard order of precedence. Note that AGREE will give a warning if you write an expression that is not linear. Some theorem provers do not reason about non-linear expressions. Non-linear integer arithmetic is undecidable and most theorem provers do not use a decidable decision procedure for non-linear real arithmetic. So it is recommended that you only use linear expressions.
- **Relation Expressions**: Relation expressions can be performed on integers or reals, but not a combination of both. Equality can be used on Booleans as well.
- **Boolean Expressions**: Boolean expressions have the standard associative properties and order of precedence.
- **Arrow Expression**: The arrow expression evaluates to the value of the expression of the left hand side of the arrow on the initial step. Otherwise it evaluates to the value of the expression on the right hand side of the arrow. The arrow expression is used with the `pre` expression to reason about past values of variables. For example, we can define a variable in an AGREE contract that starts at zero and increments by one each step in time using an equation statement:

  eq count **: int =** 0 **-> pre(** count **)** + 1

  Note that the difference between arrow expressions and the initially statements discussed earlier is very subtle. Before a subcomponents clock ticks the values of its outputs and intermediate variables defined by equation statements are only constrained by the expressions in initially statements. Once the subcomponents clock ticks for the first time the expressions in the initially statements become irrelevant and all arrow expressions in the subcomponent evaluate to the value of their left hand side expression.

While assumption and guarantee statements exclusively live in component types, component implementations exclusively contain other grammar elements. Specifically, component implementations can contain the following elements.

- **Synchrony Statement**: Synchrony statements describe the order in which the subcomponents execute. Right now we support modeling systems synchronously or quasi-synchronously. 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. The phrase `no_simult` can optionally be placed at the end of the statement to indicate that no two subcomponent clocks may tick simultaneously. This would be indicative of multiple threads scheduled to run on a single processor for example.
- **Assert Statement**: Assert statements make unchecked statements about how the component behaves. These are also used to reference variables from a subcomponent in the component contract. For the purpose of analysis assertions are treated just like the system assumptions in the formulas presented in Section 3.1. However, we never verify that the assertions actually hold. That is to say, the assertions of a subcomponent are never proven to hold like subcomponent assumptions in Formula 4. Assertions can be thought of as unchecked assumptions about the behavior of component. Assert statements can refer to equations or features defined in the component type. They are often 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. In other words, lemmas only ever appear as system guarantees on the right hand side of the implication in Formula 5.

4 Modeling the ULB with AGREE

4.1 The ULB System Component

As mentioned earlier, the FCC VSM, LOI Manager, Authentication Component, and Wescam VSM are modelled as process components in AADL. A process in AADL is thought of as an isolated memory space whereas a thread is used to describe computation that takes place within a process. We model the LOI Manager and Authentication Component as processes without any threads, but it is assumed that each process has some sort of "main thread" which performs its execution. We could model this explicitly; each process would have a single thread with the same assumptions and guarantees as the process itself. We have omitted this detail in the model.

Figure 4 shows the highest level contract of the model (contained in the ULB system component). The contract has three guarantees which are proven by the contracts of the LOI Manager, the Wescam VSM, and the Authentication Component. These guarantees are meant to capture the following English requirements:

```
--points to the decrypted stanag messages
eq auth_in : ULB_Device_Types::STANAG_4586_message.cucs_auth_req;
eq stanag_in : ULB_Device_Types::STANAG_4586_message.i;

eq loi : int;
eq override_control : bool;
eq cucs_id : int;
eq camera_commands : ULB_Device_Types::camera_command.i;
eq stanag_received : bool;

guarantee "allowed to override" : true ->
  stanag_received and
  stanag_in.m_id = 1 and
not pre(override_control) and
  auth_in.csm = 2
=> cucs_id = auth_in.cucsid;

eq message_with_loi_gt_3 : bool =
   stanag_received and stanag_in.m_id = 1 and
(auth_in.csm = 1 or auth_in.csm = 2) and auth_in.rloi > 3;

guarantee "loi greater than three always gets control" :
  message_with_loi_gt_3 and (false -> pre(loi = 3)) =>
    loi = auth_in.rloi and cucs_id = auth_in.cucsid;

eq loi_was_3 : bool = loi = 3 -> loi = 3 or pre(loi_was_3);

guarantee "if the camera moved, then loi was three": true ->
  camera_commands != pre(camera_commands) => loi_was_3; ;
```

Fig. 4. The contract in the ULB

1. If a STANAG 4586 message is received, and the CUCS is asserting to override control, and the previous CUCS did not assert control, then the new CUCS is granted control.
2. If the current CUCS in control has LOI of 3, a CUCS requesting control with a higher LOI is granted access.
3. If the camera changes state, then the LOI was 3.

```
synchrony: 5 no_simult;

eq loi_clk : bool;
eq cam_clk : bool;
assert loi_clk = loi_sw._CLK;
assert cam_clk = wescam_vsm_sw._CLK;

--grabbing variables out of subcomponents
assert loi_sw.id_in_control = cucs_id;
assert loi_sw.override_control = override_control;
assert loi_sw.loi = loi;
assert loi_sw.outside_stanag_in = stanag_in;
assert loi_sw.outside_stanag_in.m_data.cucs_auth_req = auth_in;
assert wescam_vsm_sw.camera_commands = camera_commands;
assert stanag_received = (event(loi_sw.outside_stanag_in) and loi_clk);

--wescam vsm sees all messages sent to it
assert (true -> (pre(event(wescam_vsm_sw.stanag_in)) and not pre(cam_clk))
 => (wescam_vsm_sw.stanag_in = pre(wescam_vsm_sw.stanag_in) and
event(wescam_vsm_sw.stanag_in) = true));

lemma "if the wescam gets a message, then the loi is 3" :
event(wescam_vsm_sw.stanag_in) => loi = 3;

lemma "if the loi was never 3, then an event was never sent to the wescam" :
not loi_was_3 => not wescam_vsm_sw.event_in_past;
```

Fig. 5. Assert statements in the ULB implementation

Figure 5 shows the assertions, synchrony statement, and lemmas defined in the ULB implementation. We use assert statements in the ULB implementation to refer to variables present in the ULB's subcomponents. Because AGREE only performs analysis at one level of the hierarchy at a time, this technique can be used to manually refer to information about leaf level components at higher tiers in the hierarchy. Specifically, we assert that the `stanag_received` variable defined in the contract is true if and only if the LOI Manager is currently executing and it received a STANAG message. We also assert the equivalence of

the `loi`, `cucs_id`, `override_control`, `stanag_in`, etc... variables in the LOI Manager with their corresponding variables in the ULB contract.

We assert that every STANAG message sent to the Wescam VSM is handled by the Wescam VSM before another message is sent to it. This may not be true of the actual implementation. We could possibly model some sort of queue to handle multiple messages being received by the Wescam VSM, but it would make proving properties more complicated.

The synchrony statement listed in the ULB implementation describes the order in which the subcomponents of the ULB execute. The integer argument in the statement indicates how many time a component's clock can tick in reference to the clock of every other component. In this case we somewhat arbitrarily chose 5. For example, the LOI Manager process can execute no more than 5 times since the Wescam VSM and Authentication Component have last executed. The phrase "no_simult" at the end of the synchrony statements indicates that no two clocks may tick simultaneously. This reflects the fact that all of the processes that we have modelled so far run on the same processor. The clock variables of individual components can be referred to by referencing the reserved variable "_CLK" in the subcomponent. For example, we have an assert statement saying that the variable `loi_clk` is equivalent to the clock variable of the LOI Manager.

Lemma statements are used to help the model checker prove the ULB's guarantees. Specifically, the lemma stating "If the LOI was never 3, then an event was never sent to the wescam" is used as a suggested invariant to help prove the guarantee "If the camera moved, then the LOI was three". Depending on the power of the model checker these lemmas may not be necessary to prove the guarantees, but they always have the added utility of proving more facts about the ULB.

4.2 The Authentication Component

The contract for the Authentication Component is shown in Figure 6. The component guarantees that any authorization message that it passes on to the LOI Manager 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 Manager[1].

We represented different STANAG 4586 message types by including multiple subcomponents within the STANAG 4586 message data implementation. This is shown in Figure 7. 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 Authorization Component we use the variable `auth_in` to specifically reference this portion of the STANAG 4586 message data field.

We use an initially statement to say that before the component's clock ticks it has no events being sent on its STANAG output.

[1] We list the LOI Manager last in the order of subcomponents in the ULB implementation so that it is higher in the total ordering for proving assumptions

```
eq auth_in : ULB_Device_Types::STANAG_4586_message.cucs_auth_req
   = stanag_onboard_out.m_data.cucs_auth_req;

initially:
   not event(stanag_onboard_out);

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. 6. The contract for the Authentication Component

```
data STANAG_4586_message
end STANAG_4586_message;

data implementation STANAG_4586_message.i
subcomponents
      m_id : data Base_Types::Integer;
      m_data : data STANAG_4586_message_data.i;
end STANAG_4586_message.i;

data STANAG_4586_message_data
end STANAG_4586_message_data;

data implementation STANAG_4586_message_data.i
subcomponents
      cucs_auth_req : data STANAG_4586_message.cucs_auth_req;
      payload_steer : data STANAG_4586_message.payload_steer;
end STANAG_4586_message_data.i;

data implementation STANAG_4586_message.cucs_auth_req
subcomponents
      tstamp : data Base_Types::Integer;
      vid : data Base_Types::Integer;
      cucsid : data Base_Types::Integer;
      vtype : data Base_Types::Integer;
      vsubtype : data Base_Types::Integer;
      dlid : data Base_Types::Integer;
      rloi : data Base_Types::Integer;
      cs : data Base_Types::Integer;
      csm : data Base_Types::Integer;
      wait : data Base_Types::Integer;
end STANAG_4586_message.cucs_auth_req;
```

Fig. 7. The AADL data implementation for STANAG 4586 message

4.3 The LOI Manager

The AGREE annex in the LOI Manager houses the majority of the logic in the model. The assumptions and guarantees that are present in the process were derived from the STANAG 4586 specification and from discussions with Boeing. In order to fit the annex cleanly into this document we split the contract into five pieces shown in Figures 8, 9, 10, 11, and 12. Figure 8 shows the intermediate variables defined in the annex. The LOI Manager keeps track of the current LOI, the CUCS in control, whether or not the CUCS has asserted that it is overriding control, and which control station the CUCS is controlling.

In Figure 9 we define the variable `loi_approved_for_message` to be true if the current message type is correct with respect to the LOI that has been authorized. Whether or not a specific message ID is authorized at a particular LOI level comes directly from the STANAG 4586 specification. This variable is used to determine whether or not a message is routed to either of the VSMs or if the message is ignored. In total, the LOI Manager 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 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[2].
   (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 a message is received and the current LOI is approved for the message type, then it is fowarded to the Wescam VSM if the LOI is 3 and the control station is set to the Wescam VSM[3]. Otherwise, the message is forwarded to the FCC VSM. If no message is received or if the message is not approved at the current LOI then the message is not fowarded to either VSM.

[2] Setting the LOI to zero is not explicit in the specification, but it needs to be set to some non-permissive value

[3] These restrictions about the control station are based on Boeing's requirements and not part of the STANAG 4586 specification. The specification states that camera control commands require LOI of 3

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 / readability. The LOI Manager assumes that the data fields for authorization messages are in their correct ranges. This assumption should be satisfied by guarantees from the Authentication Component.

The LOI Manager implementation provided by Boeing includes logic for searching through a list of pre-authorized CUCS to determine whether or not the requesting CUCS can obtain a certain LOI. We did not explicitly model this with our requirements. Although, it could be trivially added by creating a boolean variable that indicates whether or not the requesting CUCS was present in the list and has permissions for the requested LOI. One could even model the list in AGREE if desired. The best solution might be to just include this "CUCS is allowed requested LOI" variable in the guarantees and then make it an obligation on the component designer that lookup works correctly.

```
eq loi : int;
eq auth_in : ULB_Device_Types::STANAG_4586_message.cucs_auth_req
= outside_stanag_in.m_data.cucs_auth_req;
eq id_in_control : int;
eq override_control : bool;
eq none_in_control : bool;
eq control_station : int;
--in the actual implementation this should be ascertained
--by checking to see if the CUCS is in the auth_map with
--a specified loi
eq is_auth_loi : bool;

eq mid : int = outside_stanag_in.m_id;

eq loi2 : bool = loi = 2;
eq loi3 : bool = loi = 3;
eq loi45 : bool = loi = 4 or loi = 5;
```

Fig. 8. Part 1 of the contract for the LOI Manager

```
eq loi_approved_for_message : bool  = (
if(mid = 1) then --note that this is an auth request, so we don't route it
false
else if(mid = 20 or mid = 21) then
    loi2 or loi3 or loi45
else if(40 <= mid and mid <= 46) then
    loi45
else if(mid = 47) then
    loi3 or loi45
else if(mid = 100) then
    loi45
else if(mid = 101) then
    loi2 or loi3 or loi45
else if(102 <= mid and mid <= 108) then
    loi45
else if(200 <= mid and mid <= 206) then
    loi3
else if(mid = 207) then
    loi3 or loi45
else if(300 <= mid and mid <= 306) then
    loi2 or loi3
else if(mid = 307 or mid = 308) then
    loi3
else if(400 <= mid and mid <= 404) then
    loi2 or loi3 or loi45
else if(500 <= mid and mid <= 503) then
    loi2 or loi3 or loi45
else if(mid = 600) then
    loi3 or loi45
else if(mid = 700) then
    loi3 or loi45
else if(mid = 800) then
    loi45
else if(801 <= mid and mid <= 803) then
    loi3 or loi45
else if(mid = 804) then
    loi3
else if(mid = 805 or mid = 806) then
    loi3 or loi45
else if(mid = 900) then
    loi3 or loi45
else if(mid = 1000 or mid = 1001) then
    loi3 or loi45
else if(mid = 1100 or mid = 1101) then
    loi3 or loi45
else if(1200 <= mid and mid <= 1203) then
    loi2 or loi3 or loi45
else if(1300 <= mid and mid <= 1303) then
    loi2 or loi3 or loi45
else if(1400 <= mid and mid <= 1402) then
    loi2 or loi3 or loi45
else if(mid = 1403) then
   loi3 or loi45
else if(mid = 1500 or mid = 1501) then
    loi45
else if(mid = 1600) then
    loi45
else
false);
```


Fig. 9. Part 2 of the contract for the LOI Manager

```
eq initial_state : bool =
  loi = 0 and
  override_control = false and
  none_in_control = true and
  control_station = 0;

initially:
not event(wescam_vsm_stanag_out) and
not event(vsm_stanag_out);

initially:
 initial_state;

guarantee "No message recieved behavior":
--keep this up to date as new message types are implemented
(not event(outside_stanag_in) or outside_stanag_in.m_id != 1) => (
        initial_state ->
        loi = pre(loi) and
        override_control = pre(override_control) and
        id_in_control = pre(id_in_control) and
        none_in_control = pre(none_in_control) and
        control_station = pre(control_station)
);
```

Fig. 10. Part 3 of the contract for the LOI Manager

```
guarantee "CUCS Authorisation Requestion Behavior" :
event(outside_stanag_in) and outside_stanag_in.m_id = 1 => (
--auth request is message #1
if(auth_in.csm = 0 and auth_in.cucsid = id_in_control ) then
--relenquish control
      override_control = false and
      none_in_control = true and
      loi = 0
else if ((auth_in.csm = 1 or auth_in.csm = 2) and
prev(loi = 3,true) and
auth_in.rloi > 3) then --request control w/ greater loi
      override_control = (auth_in.csm = 2) and
      id_in_control = auth_in.cucsid and
      none_in_control = false and
      control_station = auth_in.cs and
      loi = auth_in.rloi
else if(auth_in.csm = 1 and not prev(override_control,false)) then
--request control
      override_control = false and
      id_in_control = auth_in.cucsid and
      none_in_control = false and
      control_station = auth_in.cs and
      loi = auth_in.rloi
else if(auth_in.csm = 2 and not prev(override_control,false)) then
--override control
      override_control = true and
      id_in_control = auth_in.cucsid and
      none_in_control = false and
      control_station = auth_in.cs and
      loi = auth_in.rloi
else
      initial_state ->
      override_control = pre(override_control) and
      id_in_control = pre(id_in_control) and
      none_in_control = pre(none_in_control) and
      loi = pre(loi) and
  control_station = pre(control_station)
);
```

Fig. 11. Part 4 of the contract for the LOI Manager

```
guarantee "message routing" :
if(event(outside_stanag_in) and loi_approved_for_message) then
if(control_station = 1 and loi = 3) then
   wescam_vsm_stanag_out = outside_stanag_in and
   event(wescam_vsm_stanag_out) and
   not event(vsm_stanag_out)
else
   vsm_stanag_out = outside_stanag_in and
   not event(wescam_vsm_stanag_out) and
   event(vsm_stanag_out)
else
   not event(wescam_vsm_stanag_out) and
   not event(vsm_stanag_out);

assume "valid auth data" :
   event(outside_stanag_in) =>  (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. 12. Part 5 of the contract for the LOI Manager

4.4 The Wescam VSM

The contract for the Wescam VSM process is proved by the two threads within the process: The camera thread and the STANAG thread. The guarantees from these threads prove two guarantees about the Wescam VSM.

1. If a STANAG message has never been received then the camera never moves.
2. If the camera moves, then a STANAG message was received.

These guarantees are used to prove properties in the ULB about the camera not moving unless that LOI was three. Note that there is an initially statement in the contract to constrain the values of the camera output to be zero before the Wescam VSM's clock ticks. Likewise there is a guarantee that the camera values are zero the first time the clock ticks. Constraining the camera output to be zero on the first tick even if a message is received makes it easier to phrase the two guarantees mentioned above.

The Wescam VSM's implementation just contains a synchrony statement. The value of the synchrony statement is 1, and the `no_simult` argument is used. This indicates that the two threads in the Wescam VSM run immediately after the other and never at the same time. This may or may not be accurate to how the threads are actually scheduled in the real ULB implementation.

The contract for the STANAG thread just guarantees how the STANAG payload steer message is translated to the internal camera state message. It also guarantees that it only sends a message to the camera thread if it receives a message. Likewise the camera thread just guarantees that it passes and messages it receives on its input to its output. Otherwise it keeps the camera state the same as it was on its previous execution.

5 Analyzing the Model with AGREE

AGREE runs as a plugin inside of OSATE. OSATE is an IDE for creating AADL models. The following software is needed to use AGREE.

1. The latest release of OSATE which can be found here: `http://www.aadl.info/aadl/osate/stable/`
2. The latest release of AGREE which can be found here: `https://github.com/smaccm/smaccm/releases`
3. The latest release of JKind which can be found here: `https://github.com/agacek/jkind/releases`. JKind must be added to your `PATH` environment variable for AGREE to find it. Note that Kind2 can be used remotely by changing the default AGREE preferences. JKind is not needed if Kind2 is used.
4. To use JKind, a supported SMT solver needs to be installed and added to the `PATH` environment variable. JKind supports the following SMT Solvers
   - Yices 1: `http://yices.csl.sri.com/download.shtml`
   - Z3: `http://z3.codeplex.com/`

```
initially:
not event(camera_commands) and
  camera_commands.tstamp = 0 and
  camera_commands.vid = 0 and
  camera_commands.cucsid = 0 and
  camera_commands.station_num = 0 and
  camera_commands.azimuth = 0.0 and
  camera_commands.elevation = 0.0 and
  camera_commands.hfov = 0.0 and
  camera_commands.vfov = 0.0 and
  camera_commands.hsr = 0.0 and
  camera_commands.vsr = 0.0 and
  camera_commands.lati = 0.0 and
  camera_commands.long = 0.0 and
  camera_commands.alti = 0 and
not event_in_past;

guarantee "initial state" :
(camera_commands.tstamp = 0 and
   camera_commands.vid = 0 and
   camera_commands.cucsid = 0 and
   camera_commands.station_num = 0 and
   camera_commands.azimuth = 0.0 and
   camera_commands.elevation = 0.0 and
   camera_commands.hfov = 0.0 and
   camera_commands.vfov = 0.0 and
   camera_commands.hsr = 0.0 and
   camera_commands.vsr = 0.0 and
   camera_commands.lati = 0.0 and
   camera_commands.long = 0.0 and
   camera_commands.alti = 0) -> true;

eq received_stanag_past : bool =
event(stanag_in) ->
pre(received_stanag_past) or event(stanag_in);

–if we never received a stanag message, then the camera state never changes
guarantee "no stanag means no camera" : true ->
not received_stanag_past => (camera_commands = pre(camera_commands));

eq event_in_past : bool =
event(stanag_in) -> pre(event_in_past) or event(stanag_in);

guarantee "If the camera moves, it is because it received a message": true ->
 camera_commands != pre(camera_commands) => event_in_past;
```

Fig. 13. The contract for the Wescam VSM

– CVC4: `http://cvc4.cs.nyu.edu/web/`

The solvers have different terms and conditions. So some may or may not be used for certain applications.

To install the AGREE plugins into OSATE the dropins.zip file should be extracted into the directory where OSATE is installed. Figure 14 shows the ULB AADL model in OSATE. To run AGREE, select a component implementation in the outline view on the right hand side of the screen. Then access the AGREE menu at the top of the screen (or right click the system implementation in the outline to get access to the same menu). From this menu selecting "Verify Single Layer" verifies only the contract of the system implementation that you have selected. Choosing "Verify All Layers" will run the analysis on the selected system and all components underneath it in the model hierarchy.

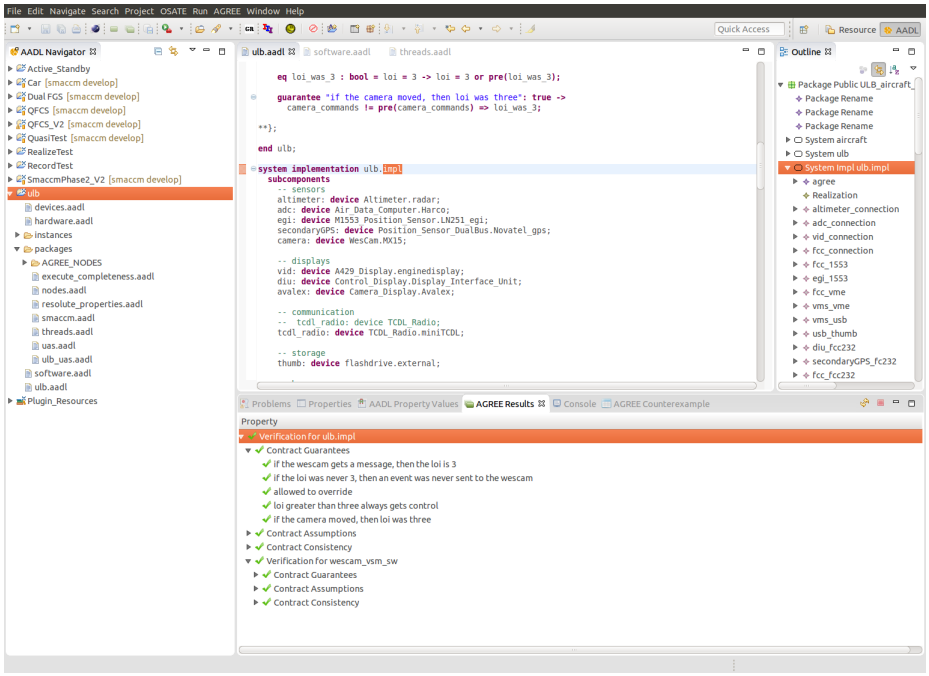

Fig. 14. The ULB AADL model in OSATE. The ULB system implementation is being selected in the outline view on the right hand side of the screen

A results view will appear at the bottom of the screen with the status of all the properties being checked. Results for each component are grouped by guarantees, assumptions, and consistency. The guarantee and assumption results are the results of checking Formulas 5 and 4, respectively. The consistency results check to see if individual contracts can be satisfied over some finite length of time. Checking consistency verifies that the guarantees that you are trying to verify are

not vacuously true. For example, if a subcomponent guarantees "false", anything can be proved in the contract of a component containing that subcomponent since it is asserted that the guarantees of a components subcomponents hold. The consistency check also verifies that the composition of the subcomponents are consistent, and that the contract being analyzed is consistent.

If a counterexample for a property is found then it will have a red icon next to it in the results dialog. Right-clicking on one of these results will bring up a menu where you can choose to view the counterexample in the console, in a spreadsheet, or in a collapsible menu. An example of the collapsible menu is shown in Figure 15.

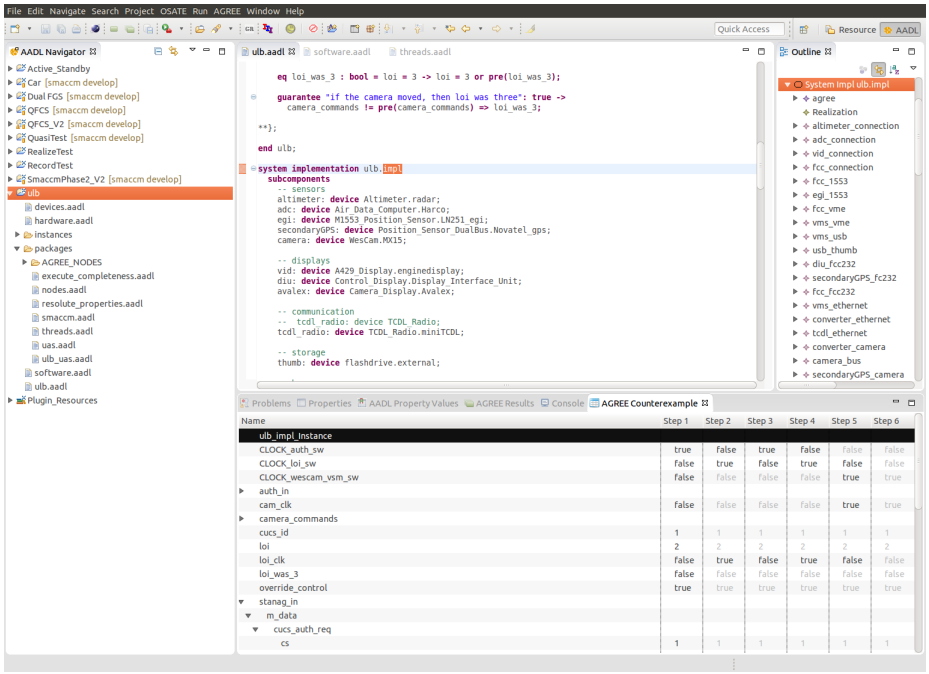

Fig. 15. A counterexample being displayed in Eclipse

There is a preferences menu accessible under Window→Preferences that can be used to change various options about AGREE's Analysis. This menu is shown in Figure 16. Here you can choose between JKind and the Kind2 model checker. If you choose JKind you can select different SMT Solvers for JKind to use. You can also decide the timeout and maximum depth for k-induction to use.

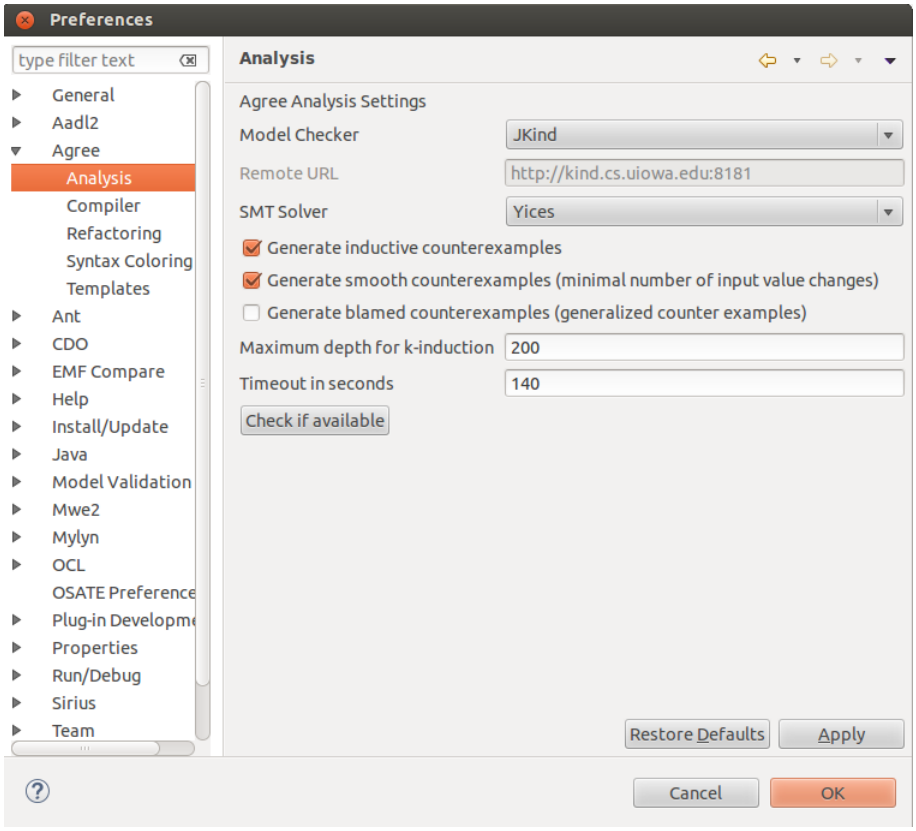

Fig. 16. The AGREE preferences menu

6 What is left to do?

Some of the properties that we have proved could be strengthened. For example, we proved that "If the camera state changes, then the LOI was three". This does not rule out a scenario where the LOI was three, then changed to two, and then the camera changes state from a CUCS authorized with an LOI of two. A stronger property to prove may be "Any message sent by a CUCS with LOI not equal to three does not change the camera state".

We have neglected to prove more "liveness-like" properties at this point because we are unsure of if the quasi-synchronous constraints we have are correct. For example, we could prove that "If a CUCS with LOI 3 sends a message to change the state of the camera, the camera state will change in n time steps".

There are many other aspects of the ULB that could be modelled with AGREE. For example, properties about how the FCC VSM handles STANAG 4586 messages could be verified. We also need to make sure that the quasi-synchronous constraints that we assumed in our analysis are accurate to the real model.

We did not model any of the messages that the VSM is obligated to return to a CUCS as specified by the STANAG 4586 protocol. If there are properties of interest with this protocol we could certainly model this.

The analysis that we have done so far assumes that the LOI Manager actually meets the requirements that we have formalized. The same goes for the requirements that we placed on the Authentication Component, Wescam VSM, and the threads within the Wescam VSM. In order for the results from AGREE to be accurate, we need to show that the implementations of these components actually meet these requirements.