Synthesizing Verified Components for Cyber Assured Systems Engineering

Eric Mercer
Department of Computer Science
Brigham Young University
Provo, Utah

Konrad Slind, Isaac Amundson, Darren Cofer
Applied Research and Technology
Collins Aerospace
Minneapolis, Minnesota

Junaid Babar, David Hardin
Applied Research and Technology
Collins Aerospace
Cedar Rapids, Iowa

***Abstract*—Cyber-physical systems, such as avionics, must be tolerant to cyber-attacks in the same way they are tolerant to random faults: they either gracefully recover or safely shut down as requirements dictate. The DARPA Cyber Assured Systems Engineering program is developing tools for design, analysis, and verification that enable systems engineers to design-in cyber-resiliency in a Model-Based Systems Engineering environment. This paper describes automated model transformations that introduce high-assurance cyber-resiliency components into a system, in particular filters and monitors that prevent malicious input and detect supply chain attacks, respectively. A formal specification defines each high-assurance component, and is used to verify that the component addresses system level cyber requirements. Implementations for these high-assurance components are directly synthesized from their specifications, and are automatically proven to preserve the exact meaning of the specifications all the way down to the binary code level. The model transformations are integrated into the Open Source AADL Tool Environment (OSATE). The paper further reports on a case study applying security-enhancing model transformations to a UAV system that uses the Air Force Research Laboratory's OpenUxAS services for route planning. In the case study, the model transformations add filters to guard against malformed input, as well as monitors to guard against ground station spoofing and malicious flight plans from OpenUxAS.**

I. Introduction

In recent years, aerospace stakeholders have realized that avionics systems are subject to possible cyber-attacks just like other cyber-physical systems. Thus, in addition to being fault-tolerant, safety-critical avionics systems must also be *cyber-resilient*. Cyber-resiliency means that the system is tolerant to cyberattacks just as safety-critical systems are tolerant to random faults: they recover and continue to execute their mission function, or safely shut down, as requirements dictate.

Unfortunately, systems engineers are currently given few development tools to help answer even basic questions about potential vulnerabilities and mitigations, and instead rely on process-oriented checklists and guidelines. Cyber vulnerabilities are often discovered during penetration testing late in the development process; or worse yet, they may be discovered only after the product has been fielded, necessitating extremely expensive and time-consuming remediation. This is not a sustainable development model.

The DARPA Cyber Assured Systems Engineering (CASE) program is targeted at developing tools for design, analysis, and verification that enable systems engineers to *design-in* cyber-resiliency for complex cyber-physical systems.[1] We have developed a Model-Based Systems Engineering (MBSE) environment called *BriefCASE* which is based on the Architecture Analysis and Design Language (AADL) [1]. BriefCASE extends the Open Source AADL Tool Environment (OSATE) to add new design, analysis, and code generation capabilities for building cyber-resilient systems.

BriefCASE provides access to two analysis tools (GearCASE [2] and DCRYPPS [3]) that can examine AADL models for potential cyber vulnerabilities and suggest cyber-security requirements to mitigate them. A library of architectural transforms guides systems engineers through automated model transformations that modify the architecture to address these requirements, possibly inserting new high-assurance components into the system. Implementations for these new high-assurance components are synthesized from formal specifications using the Semantic Properties for Language and Automata Theory (SPLAT) tool [4], [5]. Formal verification that the transformed system model satisfies its cyber requirements is accomplished via the Assume Guarantee Reasoning Environment (AGREE) [6]. AGREE is a *compositional assume-guarantee* style model checker for AADL models that attempts to prove properties about one layer of the architecture using properties allocated to its subcomponents. Cyber-resilient code implementing the verified model is then automatically generated using the High Assurance Modeling and Rapid Engineering for Embedded Systems (HAMR) toolkit [7]. If desired, this code can be targeted to the formally verified seL4 secure microkernel [8].

The two automatic transforms discussed in this paper are (1) the insertion of a filter to prevent malformed data from a malicious actor from being propagated to downstream components, and (2) the insertion of a monitor to detect (and alert) unexpected behaviors arising from untrusted components. These transformations not only change the architecture of the model by adding in new components; they also generate a formal specification for the behavior of the inserted high-assurance components in the AGREE language. Those specifications are sufficient for model checking to prove that with the incorpora-

[1]This work was funded in part by the Defense Advanced Research Projects Agency (DARPA). The views expressed are those of the authors and do not reflect the official policy or position of DARPA or the U.S. Government.

tion of these high-assurance components, the hardened system meets its cyber-resiliency requirements.

Another novel aspect of the approach is the synthesis of the AGREE specifications for the high-assurance components to CakeML, a verified compiler implementation for the functional programming language ML [9]. This paper describes in detail the synthesis path from specifications to CakeML code, providing a formal framework in which to argue correctness. CakeML then provides a verified compilation path to several different target binaries proving that the meaning of the CakeML is exactly preserved in the final binaries. Assuming that the execution schedule of the deployed cyber-hardened system is as intended by the AADL model, and that the HAMR-generated communication fabric delivers messages between components as expected, the AGREE model checking results hold for the deployed system, *i.e.*, it detects and prevents the indicated cyber-vulnerabilities over all possible finite inputs. Work is ongoing to lift this result to infinite input traces as these systems are inherently reactive and intended to run forever [10], [11].

The approach is motivated, and illustrated, in a simple example in Section II. Contract specification in the AGREE language is presented in Section III, followed by a description of the synthesis pathway in Section IV. A case study applying these transformations to an Unmanned Aerial Vehicle (UAV) system that uses the Air Force Research Laboratory's OpenUxAS services for route planning is presented in Section V. Here the transforms add filters to guard against malformed input and monitors to guard against ground station spoofing and malicious flight plans from OpenUxAS. The case study system is significantly more complex than the simple example and shows the viability of the modeling approach to a full-scale industrial design.

The BriefCASE tools are open source and publicly available [12], as is the motivating example [13], and the full UxAS model with its deployment in seL4 [14], [15]. Videos demonstrating the use of the BriefCASE tools to build the UAV example presented in Section V are also available [16].

II. Simple Example

Fig. 1 is an AADL architectural model of a software system (SW) for route planning and automated control of a UAV. It is loosely based on the system in the case study introduced in Section V. The source for the entire model is found at [13]. The system receives an automation request that is forwarded to an untrusted third-party route planner (AI). The route planner decides the flight path of the UAV based on its current position and the requested task. The waypoint manager (WM) receives the mission command as a set of waypoints from the planner and starts the UAV flying the mission, issuing waypoints to the UAV flight controller as the UAV location changes. The waypoint manager is an *as is* legacy component.

The expected behavior of the SW system, and the components in its implementation, are modeled with AGREE contract specifications. The contracts constrain input and state properties of output for component models. AGREE performs model checking on this assume-guarantee system to hierarchically prove that the composite system obeys all contract obligations under all possible finite input streams.

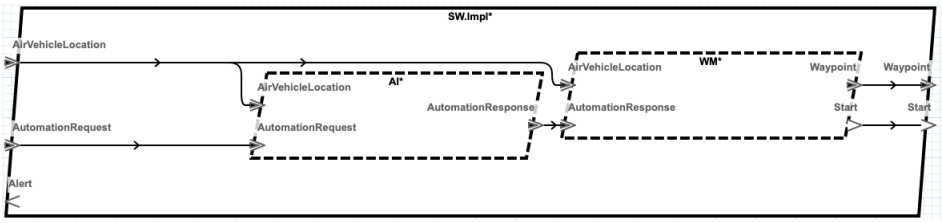
The initial AGREE contract specifications for the components and the overall system make no assumptions about the integrity of the inputs and outputs. A cyber-vulnerability analysis identifies the potential of the untrusted AI route planner component to behave maliciously. In response, the system designer modifies the AGREE specification for the AI route planner to model this ability to behave maliciously (as an untrusted component) by removing any guarantees about its output. In other words, the AI output is unconstrained in the AGREE specification, allowing it to take on any value and allowing the AI component to send that value at any time.

The contract specifications for the other components are also updated with the threat analysis information by adding in new requirements to mitigate the identified cyber-vulnerabilities. For example, a designer adds to the specification for the legacy waypoint manager assumptions about its input being *well-formed* since that is no longer known *a priori* as the AI route planner output is unconstrained. The systems designer is also responsible for defining the meaning of well-formed; in this example, it is a predicate checking that the waypoints are within bounded value ranges for latitude, longitude, and altitude, with an additional integrity check on message IDs.

The designer adds two other requirements to the SW AGREE specification related to the cyber-vulnerability analysis. The first, *Waypoint is well-formed* requires all the waypoints sent to the UAV flight controller to be well-formed (it detects if malformed waypoints are propagated downstream). The second, *Alert if start is not bounded relative to a request*, requires an automation request to correspond with an automation response either in the same step or within one step (it detects if a mission is being started without a request or if the start is delayed more than one step after a request). The goal of these two requirements is to detect if the untrusted component is trying to prevent the UAV from flying its intended mission or to fly a wrong mission.

AGREE examines the composition of the new specifications for the implementation in Fig. 1(a) to determine whether it complies with the added cyber requirements, by way of model checking. The output from the model checker is shown in Fig. 1(b). The red exclamation points designate properties that do not hold. Each of these failures comes with a counterexample. The results are not unexpected given the AI route planner's unconstrained behavior and the new assumptions about the wellformedness of the input to the legacy waypoint manager. The counterexample for *Alert if start is not bounded relative to a request* shows a response in the first time step with no matching request, a clear malicious behavior from the untrusted component.

The system implementation is cyber-hardened using BriefCASE, which automatically transforms the model by inserting high-assurance components in the form of a filter and a monitor as shown in Fig. 2(a). A filter enforces an invariant over each datum in the data stream by not forwarding input to

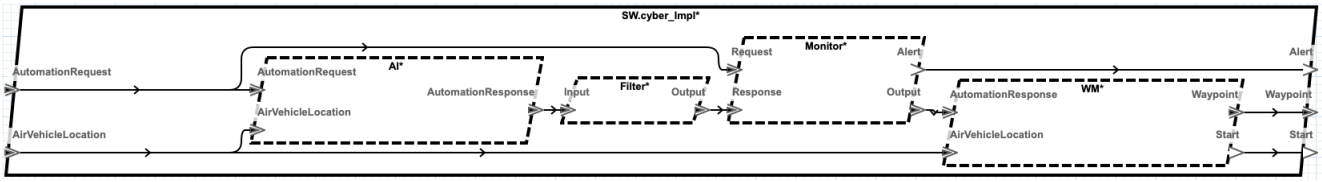


(a)

Verification for SW.Impl	6 Invalid, 7 Valid
Contract Guarantees	6 Invalid, 3 Valid
AI assume: Automation requests are well-formed	Valid (0s)
AI assume: Air vehicle location is well-formed	Valid (0s)
WM assume: Automation response is well-formed	Invalid (0s)
WM assume: Air vehicle state is well-formed	Valid (0s)
Subcomponent Assumptions	Invalid (0s)
Start includes a waypoint	Invalid (0s)
Locations required after the start waypoint	Invalid (0s)
Waypoint is well-formed	Invalid (0s)
Alert if start is not bounded relative to a request	Invalid (0s)
This component consistent	1 Valid
AI consistent	1 Valid
WM consistent	1 Valid
Component composition consistent	1 Valid

(b)

Fig. 1. Automated UAV route planning system. (a) Unhardened system. (b) Failure certificate.



(a)

Verification for SW.cyber_Impl	15 Valid
Contract Guarantees	9 Valid
AI assume: Automation requests are well-formed	Valid (1s)
AI assume: Air vehicle location is well-formed	Valid (1s)
WM assume: Automation response is well-formed	Valid (1s)
WM assume: Air vehicle state is well-formed	Valid (1s)
Subcomponent Assumptions	Valid (1s)
Start includes a waypoint	Valid (1s)
Locations required after the start waypoint	Valid (1s)
Waypoint is well-formed	Valid (1s)
Alert if start is not bounded relative to a request	Valid (1s)
This component consistent	1 Valid
AI consistent	1 Valid
WM consistent	1 Valid
Filter consistent	1 Valid
Monitor consistent	1 Valid
Component composition consistent	1 Valid

(b)

Fig. 2. Hardened UAV system. (a) The implementation with high-assurance components. (b) Passing certificate.

its output if that input violates the filter invariant. The auto-generated AGREE specification states that only well-formed inputs are passed to the output. The system developer must provide this filtering policy, but it is usually based on the existing assumptions made by downstream components that consume the filter output.

A monitor captures a relation on input data over time and is thus able to reason about temporal properties of that input. A monitor raises an alert if the specified temporal properties are ever violated. The AGREE specification for the monitor in our example states that an automation response can only be generated in conjunction with an automation request; and further, that response must come with the request or in the next step after the request. As with the filter, the system designer provides the policy, and that policy is based on the existing AGREE specification in the SW system.

The AGREE analysis of the cyber-hardened implementation with the auto-generated high-assurance components is shown in Fig. 2(b). Here AGREE provides a proof certificate that the high-assurance components guarantee the correct behavior of the SW implementation in the presence of the considered cyber vulnerabilities from the untrusted AI route planner.

The high-assurance components are automatically synthesized by SPLAT from the AGREE specifications to equivalent models in the CakeML language. CakeML itself provides a complete verified compilation to binaries for several different platforms meaning that the resulting binaries exactly preserve the meaning of the original CakeML code [9].

A similar proof is given for the synthesis of the contract model for a high-assurance component to CakeML. A high-assurance component contract has a precise meaning in terms of data streams, and the synthesis exactly preserves that meaning in the generated code. In other words, for any set of input streams that meet the component's contract assumptions,

```
eq req : bool = event(AutomationRequest);
eq avl : bool = event(AirVehicleLocation);
eq wp : bool = event(Waypoint);
eq rsp: bool = event(Start);
eq alrt : bool = event(Alert);

assume "Automation request is well-formed" :
   req => WELL_FORMED_AUTOMATION_REQUEST(AutomationRequest);
assume "Air vehicle location is well-formed" :
   avl => WELL_FORMED_WAYPOINT(AirVehicleLocation);

eq current : bool = (req = rsp);
eq previous : bool = (req and not rsp) ->
                     pre(req and not rsp) and (not req and rsp);
eq policy : bool = current or previous;
eq since : bool = alrt or (alrt and (false -> pre(since)));

guarantee "Start includes a waypoint" :
   rsp => wp;
guarantee "Locations required after the start waypoint" :
   (wp and not rsp) => avl;
guarantee "Waypoint is well-formed" :
   wp => WELL_FORMED_WAYPOINT(Waypoint);
guarantee "Alert if start is not bounded relative to a request" :
   policy or since;
```

Fig. 3. The SW component contract.

the output streams produced from the synthesized CakeML exactly match the output streams from the high-assurance component's contract.

Preserving the input/output relationship of streams between the two models lifts the contract verification results to the deployed system. If the contract model verification succeeds, then the meaning of those results hold for the deployed system, given that all other components implement their contracts, an appropriate schedule exists that follows the dependent data-flow, and the communication fabric works as expected.

III. AGREE Contract Specification

The goal of this section is to illustrate in more detail the process a system designer follows to add cyber requirements to the AGREE specifications and then automatically transform the system to insert the high-assurance components. The AGREE specifications generated by the transforms for the high-assurance components are also explained. The section ends with a concise formal statement of the meaning of an AGREE specification for a high-assurance component. This meaning is what must be preserved by the synthesis.

The AGREE specification for the SW component in the example of Section II with the added cyber requirements is given in Fig. 3. The AGREE specification language is a first-order predicate calculus that uses stream concepts, and operators, from the Lustre language [17]. As with Lustre, the semantics are synchronous data-flow where the inputs, outputs, and expressions are characterized by data streams that comply with the input assumptions. The semantics are such that the contracts are evaluated in dependency order with inputs being propagated to outputs through all the contracts until they stabilize; as such, the contracts, and thereby the top-level model, must be acyclic. Once the contracts have stabilized, the model takes a synchronous step to the next input data in the stream. The semantics do not model computation or communication delay. The output of one contract is seen at the input of any downstream contract in the same step of the input data stream.

The AGREE model checker attempts to prove several properties of the top-level model being verified. The first is that the output guarantees of each component implementing the system are strong enough to validate the input assumptions of any downstream component as well as to satisfy the guarantees of the output of the top-level component being verified (i.e., the system composition meets input assumptions at each input as well as the guarantees on the system output). These properties are reported in the expanded lists in Fig. 1(b) and Fig. 2(b). The next set of properties prove that the contract specifications for each component are self-consistent (i.e, a contract does not contradict itself). These are the unexpanded results at the bottom of the figures.

Returning back to the contract in Fig. 3, it uses `eq` statements to define variables local to the contract specification. For example, the `req` variable is equivalent to the `event(AutomationRequest)` expression. In the AGREE semantics, there is an implicit *event* input (or output) associated with every named event port in a component. The semantics used here do not buffer these events so the implicit input (or output) is only a boolean value. An `event` expression refers to that implicit input (or output) and is true when data is placed on the named port. The system contract here states assumptions on well-formed input, followed by guarantees on properties about the output.

The *Alert if start is not bounded relative to a request* guarantee is an invariant on the expression `policy or since`, meaning that either the policy holds or the alert is sounding. The `policy` is defined by two local values: `current` and `previous`. The `current` value is asserted when in the current time step there is a request with a response, or there is no request and no response.

The value of `previous` in the current time step relies on values from the previous time step. The `->` operator designates initialization, as the previous time step is undefined in the first step of the system. The left operand to the operator is the initial value of `previous` at start, which in this example is `(req and not rsp)`, because seeing a request with no response is inconclusive in the first step of the system. The right operand is the value of `previous` after the initial step. Here the `pre` operator refers to the value of the expression `(req and not rsp)` in the prior time step, `previous` is true if the previous time step made a request without a matching response and the current time step has the matching response to that request with no new request.

The value of `since` in *Alert if start is not bounded relative to a request* relies on its own value in the previous time step. The intuitive reading of the expression is that the alert has been true since the time when it first sounded. The first `alrt` sets `since` to true, and once the value of `since` is true, that value persists as long as `alrt` holds. The *Alert if start is not bounded relative to a request* guarantee defines one requirement of a cyber-hardened system implementation. Together with the other guarantees, the contract models the expected input and output of the system as a whole.

As noted previously, the original system fails to guarantee

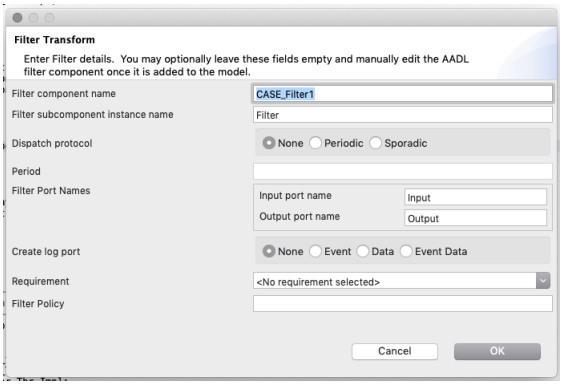


Fig. 4. Wizard for automatically transforming the model with a filter.

```
  eq policy : bool =
    WELL_FORMED_AUTOMATION_RESPONSE(Input);
  guarantee Filter_Output "Filter output is well-formed" :
    if event(Input) and policy then
      event(Output) and Output = Input
    else not event(Output);
```

(a)

```
const is_latched : bool =
  Get_Property(this, CASE_Properties::Monitor_Latched);
eq rsp : bool = event(Response);
eq req : bool = event(Request);
eq current : bool = (req = rsp);
eq previous : bool = (req and not rsp) ->
                pre(req and not rsp) and (not req and rsp);
eq policy : bool = current or previous;
eq alert : bool = (not policy)
          -> ((is_latched and pre(alert)) or not policy);
guarantee Monitor_Alert
  "Alert port tracks alert variable" :
  event(Alert) = alert;
guarantee Monitor_Output
  "Output if not alerted" :
  if event(Alert) then (not event(Output)) else
  if event(Response) then (event(Output) and (Output = Response))
  else (not event(Output));
```

(b)

Fig. 5. High-assurance component contracts. (a) The filter. (b) The monitor.

the cyber requirements. BriefCASE provides two transformations to address the failing requirements: inserting a filter and inserting a monitor. The component is added by selecting the connection in the model where the high-assurance component is to be added, and then choosing the appropriate transformation. The system designer can provide transform configuration parameters in a wizard, as shown in Fig. 4. The policy of the high-assurance component can be stated directly in the wizard, or it can be left blank. In this example, the policy is specified as `WELL_FORMED_AUTOMATION_RESPONSE(Input)`. Additionally, because a transformation is ultimately driven by a cyber requirement, BriefCASE updates an embedded Resolute assurance case [18]. Resolute keeps track of the evidential artifacts necessary for supporting the requirement, and can be run at any time to determine whether those artifacts are valid.

The AGREE contract specification generated by the transform is shown in Fig. 5(a). The guarantee is stylized for synthesis and completely defines the meaning of the output under every possible input. The resulting AGREE specification for the monitor in this example is shown in Fig. 5(b). The `is_latched` value makes the alert persistent, meaning that once the alert is raised, it is always raised. This behavior is one of the several options available in the dialogue. The definition for `policy` is taken by the system developer from the contract in Fig. 3. As before, the guarantees for the outputs are autogenerated by the tool and completely define each output under every possible input.

A. Brief Semantics Definition

Here the formal semantics of the AGREE contract specification are briefly presented to make clear the meaning of a high-assurance component. These semantics are used in the next section to argue that the synthesis preserves the same input to output behavior as the AGREE specification.

Assume that all data is in its raw form which is a contiguous sequence of bytes representing exactly what is sent over a wire by the communication fabric, so a datum is given by a string. This assumption is important to the correctness argument in the next section. An *environment*, $\theta : \mathit{lval} \mapsto \mathsf{string}$, binds *L-values* to strings where an L-value is anything that can appear on the left-hand side of an assignment such as a named port, a field in a record, a entry in an array, a local value defined by an eq-statement etc.

Let s be an AGREE contract specification for some high-assurance component. The notation θ_s is used to denote the environment that contains a binding for any L-value in the scope of s and nothing else, and the notation Θ_s denotes the universe of all such environments. The semantics of s are defined over streams, $\pi = \theta_1, \theta_2, \ldots$, which are finite sequences of environments, $\pi \in \Theta_s^*$.

The function $\mathsf{eval}\ s\ \pi$ evaluates s on the stream, π, and returns true if s is invariant along the entire stream and false otherwise. A guarantee $\mathcal{G}$ in s is *invariant* if $\mathcal{G}$ is true for each prefix of π, while an eq-statement in s is invariant if its binding in the context of every step is equivalent to the computed value of its associated expression in that same context. All guarantees and eq-statements must be invariant in the stream for the function to return true.

The meaning of an AGREE contract specification is now defined as the set of environment streams on which it is invariant.

$$\mathcal{L}(s) = \{\pi \in \Theta_s^* \mid \mathsf{eval}\ s\ \pi = \mathsf{true}\}$$

Intuitively, any stream $\pi \in \mathcal{L}(s)$, at each step, binds the L-values in the eq-statements in a way that is consistent with their associated expressions and the guarantees are all true in that same step.

We claim that a synthesized high-assurance component preserves the input to output relationship in the specification s over every stream in $\mathcal{L}(s)$. Let $\pi' = \mathsf{SynthEval}\ s\ \pi$ denote a function that synthesizes s and then evaluates that synthesized component on the stream π to create a new stream π' containing added output and other bindings. We say that two streams are equivalent in regards to a specification, denoted as $\pi =_s \pi'$, if and only if the two streams are the same length and agree on bindings for the input and output for s at every

step of the streams. We now formally state the correctness claim for synthesis.

$$\forall \pi \in \mathcal{L}(s).\ (\mathsf{SynthEval}\ s\ I_s(\pi)) =_s \pi$$

where $I_s(\pi)$ returns the corresponding stream that only retains bindings for inputs in each step and nothing else. The claim is that the synthesized component generates the same output stream as the specification for any stream belonging to the specification that is restricted to just input bindings at each step.

IV. Synthesis

Synthesis maps from model and specifications to code. The synthesis algorithm traverses the system architecture looking for occurrences of filter and monitor specifications; for each such occurrence it generates a CakeML program. In the following, we examine both filter and monitor synthesis. The latter is typically much more involved, and we will therefore devote more attention to it.

A. Filter Generation

A filter is intended to be simple, although it may make deep semantic checks. A filter has one input port and one output; messages on the input that the filter policy admits pass unchanged to the output port; all others are dropped (not passed on). We have investigated two kinds of filter. In the first, a relatively shallow scan of the input suffices to enforce the policy. For example, we have used the expressive power of Contiguity Types [19] to enforce *lightweight* bounds constraints on GPS coordinates in UxAS messages. On the other hand, a filter may need to parse the input buffer into a data structure specified in AGREE and apply a user-defined *wellformedness* property, also specified in AGREE, to the data. Arbitrarily complex wellformedness checks can be made in this way. Fig. 6 shows a combination where the checking specified by `WELL_FORMED_AUTOMATION_RESPONSE` depends on an underlying check specified by the contiguity type checking bounds on waypoints.

The verdict of a filter is made and performed within one thread invocation. Thus, in its given time slice, the following steps must be completed:

1) The filter checks to see if there is any input available. If there is none then it yields control; otherwise:
2) The input is read (and parsed if need be);
3) The wellformedness predicate is evaluated on the input;
4) If the predicate returns `true` then the input buffer is copied to the output, otherwise no action is taken; and
5) The filter yields control.

Remark 1 (Partiality). Partiality is an important consideration: steps 2 and 3 above can fail; the data might not be parseable or the wellformedness computation could be badly written and fail at runtime. In such cases, the filter should recover and yield control without passing the input onwards. In these cases, the filter is behaving as it should, but we must also guard against situations in which a *correctly specified* filter fails at runtime. This kind of defect arises when the filter *ought* to accept a message, but lack of resources results in the filter failing to do so. For example, the parse of a message might need more space than has been allocated; another example could be if the time slice provided by the scheduler is too short for the wellformedness computation to finish. Thus resource bounds need to be included in the correctness argument.

```
Waypoint =
  {Latitude : f64
   Longitude : f64
   Altitude : f32
   Check   : Assert
    (~90.0 <= Latitude and Latitude <= 90.0 andp
     ~180.0 <= Longitude and Longitude <= 180.0 and
     1000.0 <= Altitude and Altitude <= 15000.0)}

AutomationResponse =
  {TaskID : i64
   Length : u8
   Waypoints : Waypoint [3]}

fun WELL_FORMED_AUTOMATION_RESPONSE(aresp) =
  (forall wpt in aresp.Waypoints, WELL_FORMED_WAYPOINT(wpt))
  and ... ;
```

Fig. 6. Filter specification.

```
fun filter_step () =
 let val () = Utils.clear_buf buffer
     val () = API.callFFI "get_input" "" buffer
 in
   if WELL_FORMED_AUTOMATION_RESPONSE buffer
   then
     API.callFFI "put_output" buffer Utils.emptybuf
   else print"Filter rejects message.\n"
end
```

Fig. 7. Synthesized CakeML for the filter.

The contiguity type specification and wellformedness predicate for the filter are shown in Fig. 6 and the synthesized CakeML code is in Fig. 7. The code is called at dispatch by the scheduler. The `API.callFFI` is the link to the communication fabric to capture input and provide output. The body of the function restates the filter contract to make the appropriate assignments in a way that matches the truth value of the predicate in the filter guarantee. The auto-generated AGREE specification raises an alert output when the relation is violated.

B. Monitor Generation

Monitors are intended to track and analyze the externally visible behavior of system components through time. Therefore, they require more extensive computational ability than filters. In particular, our basic notion of a monitor is that it embodies a predicate over its input and output streams, and is able to access the value of a stream at any earlier point in time, if necessary. Monitors commonly use state to keep track of earlier values, unlike filters which, for us, are typically stateless components. (However, there is nothing in our approach that forbids stateful filters: they can be realized by monitors.) A monitor specification is mapped by code generation to a state transformation function of the following abstract type:

$$\mathsf{stepFn} : input \times stateVars \rightarrow stateVars \times output$$

The system scheduler *activates* components in some order. It is an obligation on the system that the scheduler follows some sensible partial order of component activation and allows each component sufficient time for its computation. Activating a monitor component takes the form of the following pseudo-code, in which the monitor evaluates the stepFn on its current inputs and the current values of the state variables, returning the new state and the output values.

$$
\begin{array}{ll}
(i_1, \ldots) & = \mathsf{readInputs}(); \\
(v_1, \ldots) & = \mathsf{readState}(); \\
(v_1', \ldots), (o_1', \ldots) & = \mathsf{stepFn}((i_1, \ldots), (v_1, \ldots)); \\
\mathsf{writeState}(v_1', \ldots); & \\
\mathsf{writeOutputs}(o_1' \ldots); &
\end{array}
$$

1) Initialization: A monitor may need to accumulate a certain minimum number of observations before being able to make a meaningful assessment of behavior. Until that threshold is attained, the monitor is essentially in its *initialization* phase. In order for correct code to be generated, monitor specifications need to spell out the values of output ports when in their initialization phases. For example, suppose a monitor does some kind of differential assessment of inputs at adjacent time slices, alerting when (say) the measured location of a UAV at times t and $t+1$ is such that the distance between the two locations is unusually large. Such a monitor needs two measurements before making its first judgement, but at the time of its first output, only one measurement will have been made. The specification must then explicitly state the correct value for the first output.

2) Step function: The stepFn works as follows:

1) Each input is parsed into data of the type specified by the port type;
2) New values for the state variables are computed, in dependency order. The discussion above on initialization now comes into play. Suppose the variable declarations have the following form:

$$
\begin{array}{l}
v_1 = i_1 \longrightarrow e_1 \\
\ldots \\
v_n = i_n \longrightarrow e_n
\end{array}
$$

   In the generated code, for the first invocation of stepFn only, the initializations are executed in order:

$$
\begin{array}{l}
v_1 = i_1; \\
\ldots \\
v_n = i_n;
\end{array}
$$

   In all subsequent steps, the *non-initialization* assignments are performed:

$$
\begin{array}{l}
v_1 = e_1; \\
\ldots \\
v_n = e_n;
\end{array}
$$

3) Values of the outputs are computed;
4) Outputs are written and the new state is written;
5) The monitor yields control.

The stepFn for the monitor of the example described in Section II is displayed in Fig. 8.

```
stepFn (Request,Response)
      (req,rsp,current,previous,policy,alert) =
let val stateVars' =
    if !initStep then
      let val req = event(Request)
          val rsp = event(Response)
          val current = (req = rsp)
          val previous = req and not(rsp)
          val policy = current or previous
          val alert = not policy
          val () = (intStep := False)
      in (req,rsp,current,previous,policy,alert)
      end
    else
      let val req = event(Request)
          val rsp = event(Response)
          val current = (req = rsp)
          val previous = pre(req and not rsp) and (not req and rsp)
          val policy = current or previous
          val alert = (is_latched and pre(alert)) or not(policy)
      in (req,rsp,current,previous,policy,alert)
      end
   val (_,rsp',_,_,_,alert') = stateVars'
   val Alert = if alert' then Some () else None
   val Output =
     if alert' then None else
     if rsp' then Some Response
     else None
in
  (stateVars', (Alert,Output))
end
```

Fig. 8. Synthesized CakeML for the monitor.

C. Component Behavior

Intuitively, for monitor specification s, stepFn is the concrete embodiment of SynthEval s, as defined in Section III-A. Its correctness amounts to showing that, given a sequence of inputs, and an initial state meeting the initialization constraints, iterating stepFn produces a π s.t. $\pi \in \mathcal{L}(s)$; and taking the union over all input sequences and initial states produces $\mathcal{L}(s)$ itself.

V. UxAS Case Study

In this section, we outline the application of the BriefCASE tool towards the development of a UAV surveillance system as a part of the DARPA CASE program. The UAV receives commands from a ground station to conduct surveillance over a geographical region. In response, the UAV's on-board mission computer generates a flight plan consisting of a series of waypoints that the UAV must traverse to complete its mission. The UAV is also given a set of *keep-in* and *keep-out* zones that may constrain its flight path.

We have modeled the system architecture of the UAV in AADL. The model includes a Mission Computer for communicating with the ground station and generating flight plans, and a Flight Control Computer for UAV navigation. The Mission Computer architecture model includes hardware components such as a processor, memory, and communication devices, as well as software. The initial software architecture model (shown in Fig. 9) contains drivers for communication with the Ground Station and Flight Control Computer, a Waypoint Manager component that provides flight plan coordinates to the Flight Control Computer, and the Flight Planner. The Flight Planner is the open-source UxAS software developed by AFRL [20].

For this application, UxAS accepts three types of messages. The *Operating Region* message defines where the UAV can

and cannot fly. The *Line Search Task* message contains a series of waypoints that the UAV should traverse. The waypoints typically lie along some geographical feature of interest, such as a river or railway. Note that the UAV may not be able to directly traverse the Line Search Task waypoints due to no-fly zone constraints specified in the Operating Region message. Anytime after receiving the Operating Region and Line Search Task messages, a Ground Station can transmit an *Automation Request* message, which instructs UxAS to generate a flight plan that satisfies these constraints. UxAS passes the flight plan in an *Automation Response* message to the Waypoint Manager. Because the Flight Control Computer can only process a small number of waypoints at a time, the Waypoint Manager parcels a small number of waypoints corresponding to the current UAV position, and sends them to the Flight Control Computer over a serial connection via the UART Driver.

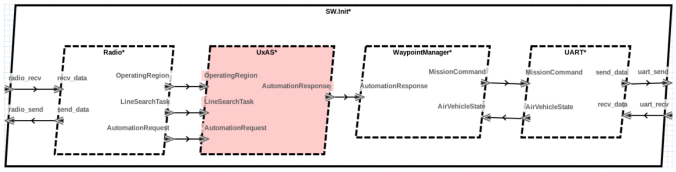


Fig. 9. Initial software architecture.

Within the software model, we have formalized some of the high-level requirements as assume-guarantee contracts. We perform a formal analysis using the AGREE tool (which is integrated with the BriefCASE environment) to verify that the model satisfies the contracts. For the initial version of our design, the verification passes. Although we are satisfied with the results of the formal verification using AGREE, we have not yet analyzed the design for cyber-vulnerabilities. In BriefCASE, we analyze the model using one (or more) of the integrated cybersecurity analysis tools. The tools generate a list of new requirements corresponding to cyber vulnerabilities found in the design, and we need to satisfy these requirements by modifying the design. For example, because we annotated the open-source UxAS component as `uncontrolled` (colored red in Fig. 9), the cyber analysis tools generate requirements for ensuring that unverified or malicious code (which could potentially be embedded in the component) will not impact other processes.

In total, seven cyber requirements are generated and imported into our model. These include four *wellformedness* requirements, two requirements for *monitoring* the behavior of the open-source UxAS component, and an *attestation* requirement for ensuring the Ground Station software has not been tampered with. Requirements are imported into the model as goals in a Resolute assurance case. Because we can run Resolute at any time during development, we can easily determine for a given snapshot of the model which requirements are not yet supported by evidence.

The intent of the *wellformedness* requirements is to prevent malformed messages from causing a buffer overrun or code injection attack. In the UAV design, such messages are most likely to originate from a remote source or the uncontrolled UxAS component. By placing filters on the connections upstream of mission-critical components, such attacks could be mitigated. The Filter transform is therefore applied for each wellformedness requirement, inserting filter components on the incoming and outgoing UxAS connections.

The filter behavior for each component is specified in AGREE. This enables formal verification within the modeling environment, and also provides a means for synthesizing the component implementation in a provably correct manner using the SPLAT tool. Because SPLAT is integrated with BriefCASE, the proof it emits when synthesizing component code is used as evidence in the Resolute goal for the corresponding mitigation. When Resolute evaluates whether such a goal is supported by evidence, it checks for the existence of the synthesis proof in addition to verifying that the architecture is correct.

The AGREE filter policies for the four UxAS connections are similar, and check that record values contained in the messages are within appropriate ranges. For example, the Automation Response message filter, which drops messages containing malformed flight plans, is defined as shown in Fig. 10. Although `Latitude`, `Longitude` and `Altitude` are defined as 64-bit floating-point values, the filter only passes messages containing waypoint values between [-90,90], [-180,180], and [0,15000], respectively.

```
-----------------------------------------
-- Automation Response message Filter --
-----------------------------------------
thread CASE_Filter_ARes
    features
        Input: in event data port CMASI::AutomationResponse.i;
        Output: out event data port CMASI::AutomationResponse.i;
    properties
        CASE_Properties::Filtering => 100;
        CASE_Properties::Component_Spec => ("Req_Filter_ARes");
        Dispatch_Protocol => Periodic;
        Period => 500ms;
        Compute_Execution_Time => 2ms .. 2ms;
        Stack_Size => CM_Property_Set::Stack_Size;
    annex agree {**

        fun WELL_FORMED_AUTOMATION_RESPONSE(msg : CMASI::AutomationResponse.i) : bool =
            forall cmd in msg.MissionCommandList,
                forall waypoint in cmd.WaypointList,
                    WELL_FORMED_WAYPOINT(waypoint);

        fun WELL_FORMED_WAYPOINT(point : CMASI::Waypoint.i) : bool =
            point.Latitude >= LATITUDE_MIN and point.Latitude <= LATITUDE_MAX and
            point.Longitude >= LONGITUDE_MIN and point.Longitude <= LONGITUDE_MAX and
            point.Altitude >= ALTITUDE_MIN and point.Altitude <= ALTITUDE_MAX;

        property CASE_Filter_policy = WELL_FORMED_AUTOMATION_RESPONSE(Input);

        guarantee Req_Filter_ARes "The filter output shall be well-formed" :
            if event(Input) and CASE_Filter_policy then
                event(Output) and Output = Input
            else
                not event(Output);
    **};
end CASE_Filter_ARes;
```

Fig. 10. Automation Response Filter specification.

In addition to monitoring the UxAS output for malformed messages, we must also monitor for suspicious behavior. This requires adding components for detecting that UxAS has crashed, as well as monitoring the correctness of the flight plans it produces. The Monitor transform is applied for this class of mitigation. In general, monitors observe a channel and compare its contents against a reference signal or constant.

The monitor policy specifies acceptable comparisons, and if violated, the monitor sends out an alert. A monitor can choose to *gate* the observed signal, in which case it also acts as a special kind of filter and drops the message if the policy is violated. The *monitoring* requirements drive two transforms. The first adds a *response* monitor to send an alert if UxAS does not emit a response within a set amount of time from receiving a request. The second adds a *geofence* monitor to ensure that generated flight plans are compliant with the specified keep-in and keep-out zones. The Geofence Monitor is a gated monitor; it prevents the observed Automation Response message from reaching the Waypoint Manager. Similar to the filters, the monitor policies are specified in AGREE. For example, the Geofence Monitor specification is shown in Fig. 11.

```
thread GeofenceMonitor
    features
        keep_in_zones: in data port CMASI::Polygon.i;
        keep_out_zones: in data port CMASI::Polygon.i;
        alert: out event port;
        observed: in event data port CMASI::AutomationResponse.i;
        output: out event data port CMASI::AutomationResponse.i;
    properties
        CASE_Properties::Monitoring => 100; -- marks this component as a monitor
        CASE_Properties::Component_Spec => ("GeofenceMonitor_alert", "GeofenceMonitor_output");
        Dispatch_Protocol => Periodic;
        Period => 500ms;
        Compute_Execution_Time => 2ms .. 2ms;
        Stack_Size => CM_Property_Set::Stack_Size;
    annex agree {**

        const is_latched : bool = true;

        property GeofenceMonitor_Policy =
              event(observed) =>
              (WAYPOINTS_IN_ZONE(GET_MISSION_COMMAND(observed), keep_in_zones)
              and WAYPOINTS_NOT_IN_ZONE(GET_MISSION_COMMAND(observed), keep_out_zones)
              and not (DUPLICATES_IN_MISSION(GET_MISSION_COMMAND(observed))));

        guarantee GeofenceMonitor_alert
         "alert trace property. To be proved by SPLAT" :
        event(alert) <=> (not GeofenceMonitor_Policy ->
                          (if is_latched then
                              Once(not (GeofenceMonitor_Policy))
                          else
                              not GeofenceMonitor_Policy
                          ));

        guarantee GeofenceMonitor_output "The monitor shall raise aan alert when" :
            if event(alert) then
              not (event(output))
            else if event(observed) then
                    (event(output) and (output = observed))
                 else
                    not (event(output));
    **};
end GeofenceMonitor;
```

Fig. 11. Geofence Monitor specification.

So far, we have addressed requirements that mitigate vulnerabilities related to malformed messages and malicious behavior *on-board* the UAV. But we also want to protect against a compromised Ground Station transmitting wellformed, but malicious commands. The final cyber requirement is mitigated by the Attestation transform [21] that adds two components to the UAV software: an Attestation Manager for evaluating remote systems like the Ground Station, and an Attestation Gate for filtering messages from sources that have not been approved by the Attestation Manager. The Attestation Manager is implemented in CakeML and automatically inserted into the application code base by BriefCASE. Because the Attestation Gate acts as a filter, the transform automatically generates its complete AGREE specification.

After transforming the model to address the cyber requirements, the software architecture now appears as shown in Fig. 12. The components in green were added to the model by way of an automated BriefCASE transform and are critical for mitigating cyber attacks. We formally verify the model with AGREE to show that all of the component contracts are satisfied, including the new contracts introduced during the model transformations. Because it is imperative that these high-assurance component implementations are correct, we run the SPLAT tool to produce provably-correct code. The synthesized code is output to a directory in the build file system with the location specified for each component in the model. The corresponding correctness proof is used in our assurance case as additional evidence that the vulnerability has been properly mitigated.

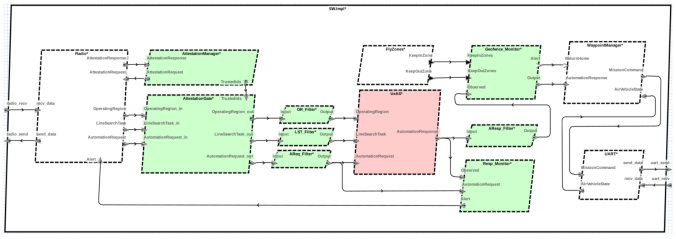


Fig. 12. Cyber-resilient software architecture.

Once we have determined that the model is correct and satisfies its cyber requirements, and the software components in the model have been implemented, either by SPLAT or other means, the system can be built and deployed. We run the HAMR tool to generate the component stubs and infrastructure code necessary to enable component communication and execution according to a specified schedule. HAMR translates the AADL system model to code that implements threading infrastructure and inter-component communication consistent with the AADL computational model. HAMR then compiles the software to execute on seL4 [8], a verified capability-based microkernel (accompanied by formal proof of spatial isolation properties down to the binary level).

The UAV and Ground Station software were deployed on ODROID XU4 hardware, and communicated with each other over ethernet. The AMASE flight simulator, representing the Flight Control Computer was run on a Linux machine and connected to the UAV ODROID via a serial connection. The UxAS implementation on the UAV was modified by adding malicious code that would prevent it from responding to Automation Requests or produce flight plans that would violate the operating region constraints. Some of the Line Search Task messages transmitted from the Ground Station contained malformed messages that would modify the UxAS behavior. In addition, we modified the Ground Station to simulate a breach for our evaluation of the attestation transform. We performed a set of tests on the initial system (Fig. 9) prior to applying our cyber-resiliency mitigations in order to verify the effectiveness of the malicious code. A third party evaluator repeated the tests on the hardened system and demonstrated that our mitigations were successful. The following scenarios were exercised (status messages from the high-assurance components in the hardened system are in Fig. 13):

a) Infected Ground Station: An application file is modified on the Ground Station, which sends the UAV on a mission-violating trajectory. On the hardened system, messages sent to the UAV were rejected by the Attestation Manager.

b) Malformed Line Search Task message: The Line Search Task message contained a waypoint with a longitude outside the permitted range to exploit vulnerabilities in the onboard software. The wellformedness filter for Line Search Task messages prevented this message from reaching the UxAS.

c) UxAS vulnerability exploit: Line Search Tasks with greater than 90 waypoints were transmitted from the Ground Station, triggering a vulnerability that crashes UxAS. When this occurs, UxAS is unable to generate an Automation Response. This vulnerability is mitigated by inserting a Response Monitor that checks to see if UxAS outputs an Automation Response message shortly after receiving an Automation Request. For this scenario, we chose for the monitor to output a status message, which would then be received by the Ground Station and an appropriate action taken.

d) UxAS trojan modifies flight plan: A trojan embedded in UxAS attempts to cause the UAV to fly into a specified keep-out zone by modifying the mission command waypoints in the Automation Response. On the hardened system the Geofence Monitor detected that it was being instructed to fly into a keep-out zone and returned the UAV to Home Base.

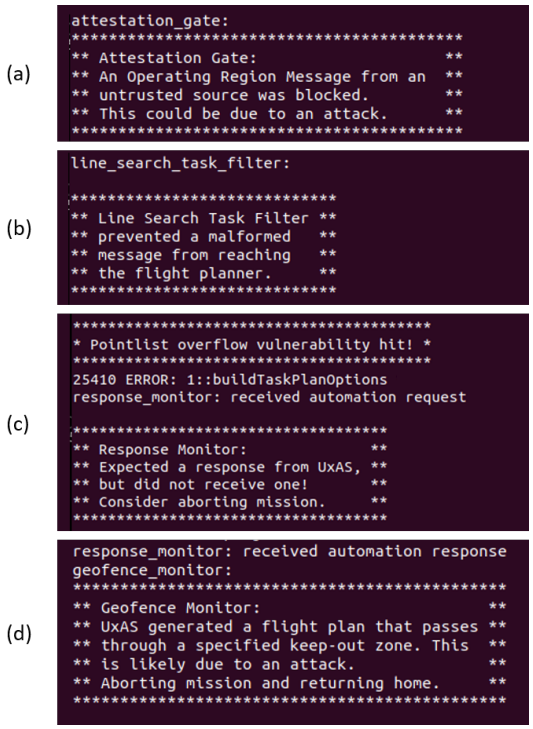


Fig. 13. Cyber-resilient system response.

VI. Related Work

Assume-guarantee reasoning for compositional verification in reactive systems is well-studied [6], [22]–[24]. Automated proofs of realizability for assume-guarantee reasoning are useful for engineers implementing components in the system [25], [26]. Algorithms for component synthesis for Lustre models using k-induction or IC3/PDR provide an automated path from the assume-guarantee reasoning to an actual satisfying node implementation [27], [28]. These algorithms synthesize code in the Lustre modeling language but do not provide a path to a low-level implementation that could be fielded.

Contracts are similar to assume-guarantee reasoning except they target programming languages. They are often more expressive than assume-guarantee reasoning being stateful and higher-order [29]. As contracts are often written in the target language, synthesizing monitors is not a problem but comes with overhead [30]. A monitor for a contract can be removed when it can be statically proven that the code preserves the contract under all possible inputs and executions [31].

Data-flow semantics are well studied [17], [32]–[35]. State machine semantics can be added to synchronous data-flow [36]. The idea is to translate imperative constructs into equivalent synchronous data flow constructs. The resulting Lustre can then be compiled. The goal is to provide a seamless connection between pure data-flow and pure control design.

There is a fully verified compiler that takes Lustre and turns it into a binary executable that is specified and verified in Coq [37]. The key is in combining infinite sequences of data-flow models with incremental manipulation of memories akin to an imperative model. CompCert is used on the backend to create the final rendered executable.

VII. Conclusion

The DARPA CASE program is creating tools for systems engineers to integrate cyber-vulnerability analysis and mitigation into their development workflow. The resulting BriefCASE tool suite includes analysis tools for generating cyber requirements, cyber resiliency tools for addressing the requirements, verification tools for ensuring design correctness, and synthesis tools for generating provably correct code. Several of the BriefCASE transforms (filter, monitor, gate) insert components into the model whose behavior can be formally specified using the AGREE language. The SPLAT tool can then automatically generate CakeML implementations for these components, along with proofs of correctness for assurance that the implementation satisfies the specification. BriefCASE was applied to a full-scale case study using the Air Force Research Laboratory's OpenUxAS software, exercising a range of built-in cyber resiliency mitigations to meet cyber-requirements. The size and scale of the study suggests BriefCASE meets the complexity demands of real-world design.

We are currently in the process of applying BriefCASE to the design of an application using the Collins Common Avionics Architecture System (CAAS) [38] on the CH-47F Chinook helicopter as part of the DARPA CASE program. Other ongoing and future work includes adding support for uninterpreted functions, mechanizing the correctness-of-synthesis proofs in HOL4, and lifting the proof results to infinite streams.

REFERENCES

[1] SAE, "Architecture Analysis and Design Language (AADL)," SAE International, Tech. Rep. AS-5506, 2009. [Online]. Available: https://www.sae.org/standards/content/as5506a/

[2] T. Patten, D. Mitchell, and C. Call, "Cyber attack grammars for risk-cost analysis," in *Proceedings of the 15th International Conference on Cyber Warfare and Security*, Norfolk, VA, 2020.

[3] R. Laddaga, P. Robertson, H. E. Shrobe, D. Cerys, P. Manghwani, and P. Meijer, "Deriving cyber-security requirements for cyber physical systems," *CoRR*, vol. abs/1901.01867, 2019. [Online]. Available: http://arxiv.org/abs/1901.01867

[4] K. Slind, "Take a SEAT: Security-Enhancing Architectural Transformations," in *Proceedings of the 20th High Confidence Software and Systems Conference*, 2020. [Online]. Available: https://cps-vo.org/hcss2020/slind

[5] D. S. Hardin and K. L. Slind, "Formal synthesis of filter components for use in security-enhancing architectural transformations," in *Proceedings of the Seventh Workshop on Language-Theoretic Security, 42nd IEEE Symposium and Workshops on Security and Privacy (LangSec 2021)*, May 2021, to appear.

[6] M. W. Whalen, A. Gacek, D. D. Cofer, A. Murugesan, M. P. E. Heimdahl, and S. Rayadurgam, "Your "what" is my "how": Iteration and hierarchy in system design," *IEEE Software*, vol. 30, no. 2, pp. 54–60, 2013. [Online]. Available: https://doi.org/10.1109/MS.2012.173

[7] *Sireum HAMR: High Assurance Modeling and Rapid Engineering for Embedded Systems*, Kansas State University, 2021. [Online]. Available: http://hamr.sireum.org/

[8] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "sel4: formal verification of an OS kernel," in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, J. N. Matthews and T. E. Anderson, Eds. ACM, 2009, pp. 207–220. [Online]. Available: https://doi.org/10.1145/1629575.1629596

[9] "CakeML," https://cakeml.org/.

[10] D. S. Hardin, K. L. Slind, J. Å. Pohjola, and M. Sproul, "Synthesis of verified architectural components for autonomy hosted on a verified microkernel," in *Proceedings of the 53rd Hawaii International Conference on System Sciences*, January 2020, pp. 6365–6374.

[11] A. Gómez-Londoño, J. Å. Pohjola, H. T. Syeda, M. O. Myreen, and Y. K. Tan, "Do you have space for dessert? a verified space cost semantics for CakeML programs," *Proc. ACM Program. Lang. (OOPSLA)*, vol. 4, pp. 204:1–204:29, 2020. [Online]. Available: https://cakeml.org/oopsla20.pdf

[12] "Formal methods workbench," https://github.com/loonwerks/formal-methods-workbench.

[13] "Synthesizing verified components for cyber assured systems engineering," https://github.com/ericmercer/verified-mon-flt-synthesis.

[14] "CASE TA6 experimental platform models," https://github.com/loonwerks/case-ta6-experimental-platform-models.

[15] "CASE TA6 platform assessemnt CAMKES apps," https://github.com/loonwerks/case-ta6-platform-assessment-camkes-apps.

[16] "CASE: Cyber Assured Systems Engineering," http://loonwerks.com/projects/case.

[17] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice, "Lustre: A declarative language for real-time programming," in *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '87. New York, NY, USA: Association for Computing Machinery, 1987, p. 178–188. [Online]. Available: https://doi.org/10.1145/41625.41641

[18] I. Amundson and D. Cofer, "Resolute assurance arguments for cyber assured systems engineering," in *Design Automation for Cyber-Physical Systems and Internet of Things (DESTION 2021)*, May 2021, to appear.

[19] K. L. Slind, "Specifying message formats with Contiguity Types," in *Proceedings of the Twelfth International Conference on Interactive Theorem Proving (ITP 2021)*, June 2021, to appear.

[20] D. B. Kingston, S. Rasmussen, and L. R. Humphrey, "Automated UAV tasks for search and surveillance," in *2016 IEEE Conference on Control Applications, CCA 2016, Buenos Aires, Argentina, September 19-22, 2016*. IEEE, 2016, pp. 1–8.

[21] A. Petz and P. Alexander, "An infrastructure for faithful execution of remote attestation protocols," in *Proceedings of the 13th NASA Formal Methods Symposium (NFM 2021)*, May 2021, to appear.

[22] D. Cofer, A. Gacek, S. Miller, M. W. Whalen, B. LaValley, and L. Sha, "Compositional verification of architectural models," in *NASA Formal Methods*, A. E. Goodloe and S. Person, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 126–140.

[23] A. Murugesan, M. W. Whalen, S. Rayadurgam, and M. P. Heimdahl, "Compositional verification of a medical device system," *Ada Lett.*, vol. 33, no. 3, p. 51–64, Nov. 2013. [Online]. Available: https://doi.org/10.1145/2658982.2527272

[24] J. Backes, D. Cofer, S. Miller, and M. W. Whalen, "Requirements analysis of a quad-redundant flight control system," in *NASA Formal Methods*, K. Havelund, G. Holzmann, and R. Joshi, Eds. Cham: Springer International Publishing, 2015, pp. 82–96.

[25] A. Gacek, A. Katis, M. W. Whalen, J. Backes, and D. Cofer, "Towards realizability checking of contracts using theories," in *NASA Formal Methods*, K. Havelund, G. Holzmann, and R. Joshi, Eds. Cham: Springer International Publishing, 2015, pp. 173–187.

[26] A. Katis, A. Gacek, and M. W. Whalen, "Machine-checked proofs for realizability checking algorithms," in *Verified Software: Theories, Tools, and Experiments*, A. Gurfinkel and S. A. Seshia, Eds. Cham: Springer International Publishing, 2016, pp. 110–123.

[27] A. Katis, G. Fedyukovich, A. Gacek, J. Backes, A. Gurfinkel, and M. W. Whalen, "Synthesis from assume-guarantee contracts using skolemized proofs of realizability," 2017.

[28] A. Katis, G. Fedyukovich, H. Guo, A. Gacek, J. Backes, A. Gurfinkel, and M. W. Whalen, "Validity-guided synthesis of reactive systems from assume-guarantee contracts," in *Tools and Algorithms for the Construction and Analysis of Systems*, D. Beyer and M. Huisman, Eds. Cham: Springer International Publishing, 2018, pp. 176–193.

[29] R. B. Findler and M. Felleisen, "Contracts for higher-order functions," *SIGPLAN Not.*, vol. 37, no. 9, p. 48–59, Sep. 2002. [Online]. Available: https://doi.org/10.1145/583852.581484

[30] C. Dimoulas, S. Tobin-Hochstadt, and M. Felleisen, "Complete monitors for behavioral contracts," in *Programming Languages and Systems*, H. Seidl, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 214–233.

[31] P. C. Nguyundefinedn, T. Gilray, S. Tobin-Hochstadt, and D. Van Horn, "Soft contract verification for higher-order stateful programs," *Proc. ACM Program. Lang.*, vol. 2, no. POPL, Dec. 2017. [Online]. Available: https://doi.org/10.1145/3158139

[32] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data flow programming language lustre," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305–1320, 1991.

[33] D. Biernacki, J.-L. Colaço, G. Hamon, and M. Pouzet, "Clock-directed modular code generation for synchronous data-flow languages," *SIGPLAN Not.*, vol. 43, no. 7, p. 121–130, Jun. 2008. [Online]. Available: https://doi.org/10.1145/1379023.1375674

[34] L. Gérard, A. Guatto, C. Pasteur, and M. Pouzet, "A modular memory optimization for synchronous data-flow languages: Application to arrays in a lustre compiler," *SIGPLAN Not.*, vol. 47, no. 5, p. 51–60, Jun. 2012. [Online]. Available: https://doi.org/10.1145/2345141.2248426

[35] J.-L. Colaço and M. Pouzet, "Clocks as first class abstract types," in *Embedded Software*, R. Alur and I. Lee, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 134–155.

[36] J.-L. Colaço, B. Pagano, and M. Pouzet, "A conservative extension of synchronous data-flow with state machines," in *Proceedings of the 5th ACM International Conference on Embedded Software*, ser. EMSOFT '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 173–182. [Online]. Available: https://doi.org/10.1145/1086228.1086261

[37] T. Bourke, L. Brun, P.-E. Dagand, X. Leroy, M. Pouzet, and L. Rieg, "A formally verified compiler for Lustre," *SIGPLAN Not.*, vol. 52, no. 6, p. 586–601, Jun. 2017. [Online]. Available: https://doi.org/10.1145/3140587.3062358

[38] "Common Avionics Architecture System," https://www.collinsaerospace.com/what-we-do/Helicopters/Rotary-Wing/Common-Avionics-Architecture-System.