Verified Synthesis of Components for Cyber Assurance

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

```
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. 1. The SW component contract.

I. AGREE

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 **??** with the added cyber requirements is given in Fig. 1. The AGREE specification language is a first-order predicate calculus that uses stream concepts, and operators, from the Lustre language [1]. 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. **??**(b) and Fig. **??**(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. 1, 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 none

Fig. 2. 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. 3. High-assurance component contracts. (a) The filter. (b) The monitor.

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 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. 2. 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 [2]. 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. 3(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. 3(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. 1. 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*, $\theta : lval \mapsto \mathsf{string}$, 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.

II. Contiguity Types

The formal specification of a component, and the synthesis of that specification, relies on *contiguity types* to define the input and output data [3] . A contiguity type is a self-describing specification for messages, and its formalism has basis in formal languages. Similar to how a regular expression implies a set of words that form its associated language, so does a contiguity type specification imply a set of messages for its language where a message is a finite sequence of contiguous bytes (e.g., a string).

What makes contiguity type specification more expressive than regular expressions is that it is self-describing meaning that the contents of the message itself may determine the rest of the message. An example is the following contiguity type specification for the message format of an `AutomationResponse` generated by a flight planner for a UAV.

```
{TaskID : i64
 Length : u8
 Waypoints : Waypoint[Length]
}
```

The `Waypoints` array size depends on the value of `Length` so the actual number of bytes in the message depends on the contents of the message itself.

The type specifications may also carry meta-information about the contents of the message.

```
{Latitude : float
 lt-rng : Assert (-90 <= Latitude <= 90)
 Longitude : float
 lng-rng : Assert (-180 <= Longitute <= 180)
 Altitude : float
 a-rng : Assert (10000 <= Altitude <= 15000)
}
```

Here the specification encodes the allowed ranges for each field of the waypoint. These assumptions restrict the resulting language to include only conforming messages and can be checked while constructing a message from a sequence of bytes. The notation $\mathcal{L}_\theta(\tau)$ denotes the language defined by

$$\begin{array}{rcl}
c & = & \mathsf{input}\ [(f:\tau)\ldots] \\
 & & \mathsf{output}\ [(f:\tau)\ldots] \\
 & & \mathsf{eq}\ [(f:\tau := \mathit{exp})\ldots] \\
 & & \mathsf{guarantee}\ [(f = \mathit{exp})\ldots] \\
 & & \\
\mathit{lval} & = & f \mid \mathit{lval}\,[\mathit{exp}] \mid \mathit{lval}.f \\
 & & \\
f & = & \mathit{varName} \\
 & & \\
\mathit{exp} & = & \mathsf{Loc}\ \mathit{lval} \mid \mathsf{nLit\ nat} \mid \mathit{constname} \\
 & \mid & \mathit{exp} + \mathit{exp} \mid \mathit{exp} * \mathit{exp} \\
 & \mid & (\mathit{exp}\ \rightarrow\ \mathit{exp}) \\
 & \mid & (\mathsf{pre}\ \mathit{exp}) \\
 & \mid & (\mathsf{ite}\ \mathit{bexp}\ \mathit{exp}\ \mathit{exp}) \\
 & \mid & \mathit{bexp} \\
 & & \\
\mathit{bexp} & = & \mathsf{bLoc}\ \mathit{lval} \mid \mathsf{bLit\ bool} \mid \neg\mathit{bexp} \mid \mathit{bexp} \wedge \mathit{bexp} \\
 & \mid & \mathit{exp} = \mathit{exp} \mid \mathit{exp} < \mathit{exp}
\end{array}$$

Fig. 4. Syntax for high-assurance component specifications.

the specification τ using the environment θ for expression evaluation.

Every contiguity type specification has a corresponding CakeML *matcher* that when given a message string returns true or false if that message belongs to the language of the specification. If the message does belong to the language, an *environment* is provided to access each part of the message. An environment, $\theta : \mathit{lval} \mapsto \mathsf{string}$ binds *L-values* to strings, where an L-value is an expression that can appear on the left hand side of an assignment (e.g., `AutomationRequest.Waypoints[0].Latitude`).

The main result of contiguity types is the proof of the relationship between the language of the specification and the synthesized matcher from the specification that is summarized below.

$$\mathsf{match}\ s_1 s_2 = \mathsf{SOME}(\theta, s_2) \Rightarrow \theta(\tau) \cdot s_2 = s_1 s_2 \wedge s_1 \in \mathcal{L}_\theta(\tau)$$

If there is a match on the substring s_1, then reconstituting the string from the resulting environment and concatenating it with s_2 yields the original string, and the matched string s_1 is in the language of the type specification.

III. Semantics

The specification language for a high-assurance component is in Fig. 4. A specification defines the inputs, outputs, local values, and guarantees for each output. A type τ is a contiguity type, and $(f : \tau)\ldots$ means zero or more repetition (e.g., Kleene star). An *lval* must eventually resolve to something that can be assigned. The expression language divides out Boolean expressions to simplify the semantics but is otherwise typical. The Loc and bLoc refer to the value of an *lval*, while nLit and bLit indicate a literal. The language includes the initialization ($\rightarrow$), pre, and if-then-else (ite) operators.

Change the following paragraph and preceding paragraphs to define the specification as the core language with a well defined normal form where expressions are flat, lvals are unique etc. It takes care of all the normal semantic checks related to types, dependency order, etc. Anything in the core language has normal form and is perfect.

The semantics are only defined for *well-formed* specifications. A specification is well-formed if and only if

1) Every $lval$ is unique;
2) the eq list is in dependency order and the expressions are acyclic;
3) the associated $lval$ with each Loc and bLoc expression is a valid reference in the environment;
4) the associated literal with each nLit and bLit has the correct type;
5) pre expressions do not refer past the beginning of the associated streams;
6) the expression list from guarantee exactly corresponds in size and order to the list from output; and
7) every expression in the list from guarantee defines its corresponding output value under all input combinations.

These checks are part of the synthesis but omitted to simplify the presentation.

An environment, $\theta : lval \mapsto \mathsf{string}$ binds L-values to strings.

The well-formed assumption enables the use of a single global environment for the semantics. The semantics are synchronous data-flow on a single clock defined over a sequences of environments where θ^i is the i^{th} environment in the stream. Expression evaluation is defined in the context of this environment stream as shown is Fig. 5. Here, $\mathsf{eval}\ i\ e$ carries with it the index of the environment to be used for the expression. $\Delta : \mathsf{string} \rightarrow \mathbb{N}$ binds constant names to numbers. Functions $\mathsf{toN} : \mathsf{string} \rightarrow \mathbb{N}$ and $\mathsf{toB} : \mathsf{string} \rightarrow \mathsf{bool}$ interpret byte sequences to numbers and booleans, respectively.

Each environment in the stream is initially partial meaning that it only contains mappings for the inputs. *Stepping* the specification updates the current environment and checks the invariance of the guarantees. In other words, at the i^{th} step, θ^i is updated with the result of the sequential evaluation of the eq-statements in the specification and then the guarantees are checked for invariance.

The notation $(lval \mapsto slice) \bullet \theta$ denotes the addition of binding $lval \mapsto slice$ to θ. Create an eval function for eq-statement. Map it to the list of statements. Create a similar function for the guarantees with its map that fails if invariance does not hold. Define the language of the specification using the contiguity notation. Need to munge all the types into a single τ, but the gist is the language is any finite stream possible that conform to the input specification are the result of the eq-statements, and are invariant. The set is prefix closed (trace theory).

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 [3] 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.

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

$$\mathsf{eval}\ i\ e = \mathsf{case}\ e \begin{cases} \mathsf{Loc}\ lval & \Rightarrow \quad \mathsf{toN}(\theta^i(lval)) \\ \mathsf{nLit}\ n & \Rightarrow \quad n \\ constname & \Rightarrow \quad \Delta(constname) \\ e_1 + e_2 & \Rightarrow \quad \mathsf{eval}\ i\ e_1 + \mathsf{eval}\ i\ e_2 \\ e_1 * e_2 & \Rightarrow \quad \mathsf{eval}\ i\ e_1 * \mathsf{eval}\ i\ e_2 \\ e_1 \to e_2 & \Rightarrow \quad \mathbf{if}\ i = 0\ \mathbf{then}\ \mathsf{eval}\ i\ e_1\ \mathbf{else}\ \mathsf{eval}\ i\ e_2 \\ (\mathsf{pre}\ e) & \Rightarrow \quad \mathsf{eval}\ i - 1\ e \end{cases}$$

$$\mathsf{evalB}\ i\ b = \mathsf{case}\ b \begin{cases} \mathsf{bLoc}\ lval & \Rightarrow \quad \mathsf{toB}(\theta^i(lval)) \\ \mathsf{bLit}\ b & \Rightarrow \quad b \\ \neg b & \Rightarrow \quad \neg(\mathsf{evalB}\ b) \\ b_1 \lor b_2 & \Rightarrow \quad \mathsf{evalB}\ i\ b_1 \lor \mathsf{evalB}\ i\ b_2 \\ b_1 \land b_2 & \Rightarrow \quad \mathsf{evalB}\ i\ b_1 \land \mathsf{evalB}\ i\ b_2 \\ e_1 = e_2 & \Rightarrow \quad \mathsf{eval}\ e_1 = \mathsf{eval}\ i\ e_2 \\ e_1 < e_2 & \Rightarrow \quad \mathsf{eval}\ e_1 < \mathsf{eval}\ i\ e_2 \end{cases}$$

Fig. 5. Expression evaluation in the context of a stream on environments.

```
 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.

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:

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

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.

```
$(i_1, \ldots)$                         $= \mathsf{readInputs}();$
$(v_1, \ldots)$                         $= \mathsf{readState}();$
$(v_1', \ldots), (o_1', \ldots)$        $= \mathsf{stepFn}((i_1, \ldots), (v_1, \ldots));$
$\mathsf{writeState}(v_1', \ldots);$
$\mathsf{writeOutputs}(o_1' \ldots);$
```

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

```
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.

now comes into play. Suppose the variable declarations have the following form:

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

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

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

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

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

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 **??** is displayed in Fig. 8.

C. Component Behavior

Intuitively, for monitor specification s, stepFn is the concrete embodiment of SynthEval s, as defined in Section I-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.

References

[1] 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. 178188. [Online]. Available: https://doi.org/10.1145/41625.41641

[2] 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.

[3] 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.