Synthesizing Verified Cyber Assured Components

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

I. Contiguity Types

The formal specification of a component, and the synthesis of that specification, relies on *contiguity types* [1] to define the input and output data (cite contiguity). A contiguity type is a self-describing specification for messages. Its formalism has basis in formal languages. Similar to how a regular expression implies a set of words that form its 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 `AutomationResponse` from the system in the previous section with its contiguity type specification.

```
{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 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, $\theta : lval \mapsto \mathsf{string}$ 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`).

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

Fig. 1. Syntax for high-assurance component specifications.

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.

II. Semantics

The specification language for a high-assurance component is in Fig. 1. 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. 2. Here, 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 toN : string $\rightarrow \mathbb{N}$ and toB : string $\rightarrow$ 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).

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

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

Fig. 2. Expression evaluation in the context of a stream on environments.

References

[1] 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.