Copland JSON Exchange Format

Adam Petz

Institute for Information Sciences
The University of Kansas
Lawrence, KS 66045
{ampetz}@ku.edu

1 Copland Phrase and Evidence Grammars

$$
\begin{aligned}
t &\leftarrow A \mid @_p\, t \mid (t \rightarrow t) \mid (t \overset{\pi}{\prec} t) \mid (t \overset{\pi}{\sim} t) \\
A &\leftarrow \mathsf{ASP}\ \bar{a} \mid \mathsf{CPY} \mid \mathsf{SIG} \mid \mathsf{HSH}
\end{aligned}
$$

Fig. 1. Copland Phrase grammar where:
$\bar{a} = (m,\ \bar{s},\ p,\ r)$; $m = \mathit{asp\_id} \in \mathbb{N}$; $\bar{s}$ is a list of string arguments; $p = \mathit{place\_id} \in \mathbb{N}$; $r = \mathit{target\_id} \in \mathbb{N}$; and $\pi = (\pi_1, \pi_2)$ is a pair of evidence splitting functions.

$$
\begin{aligned}
E_T \leftarrow\ & \mathsf{mt} \mid \mathsf{N_E}\ \mathsf{n} \mid \mathsf{ASP_E}\ \bar{a}\ p\ E_T \\
& \mid \mathsf{SIG_E}\ p\ E_T \mid \mathsf{HSH_E}\ p\ E_T \\
& \mid \mathsf{SS_E}\ E_T\ E_T \mid \mathsf{PP_E}\ E_T\ E_T
\end{aligned}
$$

Fig. 2. Evidence Type grammar where:
$\bar{a}$ and p are as in Fig. 1 and $n = \mathit{nonce\_id} \in \mathbb{N}$.

$$
\begin{aligned}
E_{Tc} \leftarrow\ & \mathsf{mt_c} \mid \mathsf{N_c}\ \mathsf{n}\ \mathsf{bs} \mid \mathsf{ASP_c}\ \bar{a}\ p\ \mathsf{bs}\ E_{Tc} \\
& \mid \mathsf{SIG_c}\ p\ \mathsf{bs}\ E_{Tc} \mid \mathsf{HSH_c}\ p\ \mathsf{bs}\ E_T \\
& \mid \mathsf{SS_E}\ E_{Tc}\ E_{Tc} \mid \mathsf{PP_E}\ E_{Tc}\ E_{Tc}
\end{aligned}
$$

Fig. 3. Typed Concrete Evidence grammar where:
$\bar{a}$, p, and n are as in Fig. 2 and $\mathsf{bs} \in \mathsf{BS}$ (binary values).

2 JSON Schemas

We represent Copland terms (t in Figure 1), Evidence Types (E_T in Figure 2), and Typed Concrete Evidence (E_{Tc} in Figure 3) as Algebraic Data Types (ADTs) in the Attestation Manager prototypes. In JSON we represent each ADT as an object with two fields:

1. constructor-the constructor name as a JSON string (< string >). Constructor names must be unique for unambiguous parsing.
2. data-An ***ordered*** JSON array (< array >) that holds the arguments for that particular constructor. Members of the data array will differ from constructor to constructor.

The general schema for ADTs (labelled by placeholder < ADT >) is as follows:

```
{
  "constructor": < string >,
  "data": < array > | < ADT >
}
```

The < ADT > alternative for the data field accounts for degenerate nesting of constructors (for example in the ASP constructors below).

2.1 Copland Phrase JSON Schemas

The following JSON object schemas correspond to the constructors of the Copland phrase grammar (t) in Figure 1, and satisfy the < term > placeholder. The < number > and < string > placeholders are for the standard json number and string datatypes. The order of items in the "data" subarray is significant–they match the order of arguments to each constructor. Finally the placeholder < SP > stands for an evidence splitter, a JSON string with one of the constant values "ALL" or "NONE".

< asp_params > := [< number >, [< string >], < number >, < number >]

```
{
  "constructor": "Coq_asp",
  "data": {"constructor": "ASPC",
           "data": < asp_params >
           }
}
```

```
{
  "constructor": "Coq_asp",
  "data": {"constructor": "CPY"}
}
```

```
{
  "constructor": "Coq_asp",
  "data": {"constructor": "SIG"}
}
```

```
{
  "constructor": "Coq_asp",
  "data": {"constructor": "HSH"}
}
```

```
{
  "constructor": "Coq_att",
  "data": [ < number >,
            < term > ]
}
```

```
{
  "constructor": "Coq_lseq",
  "data": [ < term >,
            < term > ]
}
```

< SP > := “ALL” | “NONE”

```
{
  "constructor": "Coq_bseq",
  "data": [ [< SP >, < SP >],
              < term >,
              < term > ]
}
```

```
{
  "constructor": "Coq_bpar",
  "data": [ [< SP >, < SP >],
              < term >,
              < term > ]
}
```

2.2 Copland Evidence Type Schemas

The following JSON object schemas correspond to the constructors of the Evidence Type grammar (E_T) in Figure 2, and satisfy the < evidence_type > placeholder.

```
{
  "constructor": "Coq_mt"
}
```

```
{
  "constructor": "Coq_nn",
  "data": [ < number > ]
}
```

```
{
  "constructor": "Coq_uu",
  "data": [ < asp_params >,
            < number >,
            < evidence_type > ]
}
```

```
{
  "constructor": "Coq_gg",
  "data": [ < number >,
            < evidence_type > ]
}
```

```
{
  "constructor": "Coq_hh",
  "data": [ < number >,
            < evidence_type > ]
}
```

```
{
  "constructor": "Coq_ss",
  "data": [ < evidence_type >,
            < evidence_type > ]
}
```

```
{
  "constructor": "Coq_pp",
  "data": [ < evidence_type >,
            < evidence_type > ]
}
```

2.3 Copland Typed Concrete Evidence Schemas

The following JSON object schemas correspond to the constructors of the Typed Concrete Evidence grammar (E_{T_c}) in Figure 3, and satisfy the < evidence_conc > placeholder. Note: Constructor fields that hold binary data (bs parameters in the grammar) become base64-encoded JSON strings (< b64_string >), holding arbirary binary data–hashes, nonces, signatures, etc:

< b64_string > := < string > (Base64 encoded)

```
{
  "constructor": "Coq_mtc"
}
```

```
{
  "constructor": "Coq_nnc",
  "data": [ < number >, < b64_string > ]
}
```

```
{
  "constructor": "Coq_uuc",
  "data": [ < asp_params >,
            < number >,
            < b64_string >,
            < evidence_conc > ]
}
```

```
{
  "constructor": "Coq_ggc",
  "data": [ < number >,
            < b64_string >
            < evidence_conc > ]
}
```

```
{
  "constructor": "Coq_hhc",
  "data": [ < number >,
            < b64_string >
            < evidence_type > ]
}
```

```
{
  "constructor": "Coq_ssc",
  "data": [ < evidence_conc >,
            < evidence_conc > ]
}
```

```
{
  "constructor": "Coq_ppc",
  "data": [ < evidence_conc >,
            < evidence_conc > ]
}
```

2.4 Message Schemas

$$
\begin{aligned}
RequestMessage = \{ \\
\quad toPlace &:: \mathsf{p}, \\
\quad fromPlace &:: \mathsf{p}, \\
\quad reqNameMap &:: \mathsf{p} \Rightarrow \mathsf{Address}, \\
\quad reqTerm &:: \mathsf{t}, \\
\quad reqEv &:: [\ \mathsf{bs}\ ]\ \}
\end{aligned}
$$

$$
\begin{aligned}
ResponseMessage = \{ \\
\quad respToPlace &:: \mathsf{p}, \\
\quad respFromPlace &:: \mathsf{p}, \\
\quad respEv &:: [\ \mathsf{bs}\ ]\ \}
\end{aligned}
$$

Fig. 4. Request and Response Message record structures.

We represent Request and Response Messages as record structures in Figure 4. Their respective JSON object schemas are as follows:

```
{
  "toPlace": < number >,
  "fromPlace": < number >,
  "reqNameMap": < nameMap >,
  "reqTerm": < term >,
  "reqEv": < raw_evidence >
}
```

```
{
  "respToPlace": < number >,
  "respFromPlace": < number >,
  "respEv": < raw_evidence >
}
```

where

< raw_evidence > := [< b64_string >]

< nameMap > is a JSON object of the following form:
$\{pl_1{:}addr_1, pl_2{:}addr_2, ...\}$

where pl_1, pl_2, ... are JSON key strings that represent a Copland place identifier (i.e. "1", "2", ...) and $addr_1$, $addr_2$, ... are JSON strings (< string >) that represent platform addresses. We leave address strings abstract in this specification, but an example usage would be a string of the form ip:port.

We represent Sig Request and Response Messages as record structures in Figure 5. Their respective JSON object schemas are as follows:

```
{
  "evBits": < b64_string >
}
```

```
{
  "sigBits": < b64_string >
}
```

$$SigRequestMessage = \{evBits :: \mathsf{bs}\}$$

$$SigResponseMessage = \{sigBits :: \mathsf{bs}\}$$

Fig. 5. Sig Request and Response Message record structures.

We represent Asp Request and Response Messages as record structures in Figure 6. Their respective JSON object schemas are as follows:

```
{
  "aspArgs": < asp_params >,
  "aspInputEv": < raw_evidence >
}
```

```
{
  "aspBits": < b64_string >
}
```

$$AspRequestMessage = \{ \\ aspArgs :: \bar{\mathsf{a}}, \\ aspInputEv :: [\ \mathsf{bs}\ ]\ \}$$

$$AspResponseMessage = \{aspBits :: \mathsf{bs}\}$$

Fig. 6. Asp Request and Response Message record structures.