# Unified Behavior Modeling

Peter H. Feiler

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.



DM20-0065

# Objective

To investigate a common approach to modeling flow graphs, behavior specifications, and error behavior specifications

To simplify the three layered approach of EMV2 specifications

To simplify these specifications as annex declarations

To leverage V3 unified type system and expression language

Proposal:

Common syntax with variation in expression notation

Token semantics for use in Meta behavior (error, security)

# Existing and Desired Mechanisms

AADL mode state machine

- One per component in hierarchy

AADL flow specifications

- Currently linear paths with sources and sinks
- Desired: branch/merge points

BA

- extension of mode states
- condition – transition – actions model

EMV2

- Token propagation & transformation with sources/sinks
- State-full behavior: state transformation, state-sensitive output

Agree

- Functional input to output mapping

Product line constraints

- Configuration to satisfy feature label constraint

# Representation of State

Use of enumeration type to define states of a state machine

- Reusable definition of state machine states
- Transitions tend to be context specific

State variable

- Holds the current state of a state machine
- Varname : **state** <enumeration type reference> ;
  - Could be represented by an expression language variable
- Use for mode states, error states, compute behavior states

# Modes and Mode-Specific Declarations

State variable as mode

- Use mode keyword
- Use @mode annotation

"In modes" – state condition

- **when** <state var> **in** ( <list of states> );
  - State condition can also be <state var> = <state enum literal>
  - Prefer **when** over **if** as **if** is used in **if then** expression
- Used on subcomponent, connection, property, flow spec, etc

```
system gps.i is
mode : state threestate;
locator : process sub.i1 when mode in ( s0 , s1 ) ;
end;
```

# State Transitions

State transition declaration

- Transname : **transition** <flow condition> **->** <target state> **when** <state condition>;

- Transition can apply to more than one current state
- Condition is triggered by incoming features, binding points, outgoing subcomponent features, generators

```
@EM{
    s : state threestate;
    eact1: generator  (ServiceOmission) ;
    actuate1: transition eact1 in (ServiceOmission) -> s = s1 when s in (s0) ;
```

# Generator

Component internal source of events, data, or tokens

- AADL V2 internal features
- Specify type of data, token being generated
- Properties for generation rates
- User labeled generator kind
  - E.g., use for error, recover, repair events in error mdoel

```
thread interface sense  is
    p1: out feature ;
@SEC{
    e1: error generator (Virus, DirtyWord, Classified) ;
    sense2:  token source e1 ->  p1;
};
end;
```

# Flow Specifications

Abstraction of flow through component

- Map one or more inputs to one or more outputs
- Specify flow source and sinks

```
thread interface filter is
insignal1 : in feature;
insignal2 : in feature;
outsignal : out feature;
filter2: flow (insignal1 , insignal2) ->  outsignal;
end;
```

```
thread interface actuate  is
    p1: in feature ;
    effect: out feature ;
    taction: flow  p1 -> sink;
end;
```

Flow assignment

Flow specifications can be mode specific

- Assign a path sequence as flow implementation to a flow spec

```
process interface control is
insignal: in port;
outaction: out port;
processflow: flow insignal -> outaction;
end;

process control.impl is
    dofilter: thread filter;
    docompute: thread compute;
    extin: connection insignal -> dofilter.insignal;
    ftoc: connection dofilter.outsignal -> docompute.insignal;
    extout: connection docompute.outsignal -> outaction ;
    processflow => flow dofilter.filterpath -> ftoc -> docompute.computepath ;
end;
```

Idea: connection assignment to insert a subcomponent flow

# Typed Token Behavior

## Token propagation and transformation

- Data or Meta data associated with input and output
- Simplified condition logic and output results

```
TTName : token <token condition> -> sink | <Token Results>
        [ when <state condition> ] ;
```

```
thread interface sense  is
    p1: out feature ;
@SEC{
    e1: error generator (Virus, DirtyWord, Classified) ;
    sense2:  token source e1 ->  p1;
};
end;
```

## Token results

- Comma separate list of token output or detection event
  - Token output: outp = <type reference>
  - Detection output: outevp!  [ **(** value ) ]

BA syntax

## Types of tokens

- Reference to user defined data types

# Token Condition

Input constraint as condition element

- input token contained in specified set
- Input1 **in (** <list of types> **)**

Condition expression

- Multi-literal operation of condition elements
- Operators: any, all, one of, <k> of, <k> ormore, <k> orless

```
thread interface filter is
insignal1 : in feature;
insignal2 : in feature;
outsignal : out feature;
@EM{
    filter2: token all (insignal1 in (ServiceOmission), insignal2  in (ServiceOmission))
      |->  outsignal=ServiceOmission;
};
end;
```

# Semantics of Token Behavior

## Transformation

- matching input condition to specified output value -> new token value

## Condition expression without explicit type

- Any incoming type

## Token out without explicit type

- All incoming tokens as set of tokens
- Any, one of: single output token value

## No token specification

- Interpret flow specification if present
- All inputs to all output mapping (default)

# Use Cases

System models with token sources and sinks only

- This is the John Hatcliff virus/dirty word example
- Add token source and sink specifications to model
  - For sources a generator declaration is sufficient if applicable to all outputs

EMV2 like behavior supporting unhandled error types

- Including FTA and fault impact analysis
- EMV2 FTA and impact analysis assume complete specifications
  - Indicate but do not propagate unhandled token values

# Annotations

Ability to tag AADL declarations with annotation tag

- User chosen tag ID
- Annotation with name/value parameters
- Individual declarations or annotation blocks around multiple declarations

Interpreted by generation and analysis tools

- @EM annotation for error model specifications
- @SEC annotation for security related typed token system
- Consistency rule: token type references must be to tokens with the same annotation tag

# Security Token Example

```
package SimpleTokenPropagationExample is

    @SEC type Virus;
    @SEC type DirtyWord;
    @SEC type Classified;

    interface actuator is
        inp: in feature;
        effect: out feature;
    end;
    interface sensor is
        outp: out feature;
        fea: feature;
    @SEC{
        e1: generator (Virus, DirtyWord, Classified) ;
    };
    end;

    interface control is
    insignal1: in feature;
    insignal2: in feature;
    outaction: out feature;
    @SEC{
        filter2: token insignal1 in (Virus,Classified) ->  sink;
    };
    end;


    system interface conntop is
        effect : out feature;
    end;

    system conntop.i is
        sense1: abstract sensor;
        sense2: abstract sensor;
        processing: process control;
        actuate: abstract actuator;
        sensetocontrol1: connection sense1.outp -> processing.insignal1;
        sensetocontrol2: connection sense2.outp -> processing.insignal2;
        controltoactuate: connection processing.outaction -> actuate.inp;
        effectprop: connection actuate.effect -> effect;

        effect1: flow actuate.effect  ->  effect;
    end;
```

# Representing EMV2

Error propagations/flows

- Error source, sink, path
- Now we allow logic expressions on incoming

Component error behavior

- We integrate it with error flow into a single specification
- Use state variable to represent error state machine
- Error state transition expressed by state transition on error state variable
- Outgoing propagation condition: token flow that may be conditional on error state
- Error detection condition: token flow with detection output as result

Composite error behavior

- Condition on subcomponent states resulting in composite target state

# EMV2 Equivalent Example

```
thread interface sense  is
    p1: out feature ;
@EM{
    s: state threestate;
    e1: error generator (ServiceOmission, cycles) ;
    sense2:  token e1 -> p1 ;
};
end;

thread interface actuate  is
    p1: in feature ;
    effect: out feature ;
    taction: flow  p1 -> sink;
@EM{
    s : state threestate;
    eact1: generator  (ServiceOmission) ;
    actuate1: transition eact1 in (ServiceOmission) -> s = s1 when s in (s0) ;
    actuate2: token p1 in (ServiceOmission) -> sink when s in (s0);
    actuate3: token p1 in (ServiceOmission) -> effect=ServiceOmission when s in( s1);
};
end;

thread interface filter is
insignal1 : in feature;
insignal2 : in feature;
outsignal : out feature;
@EM{
    filter2: token all (insignal1 in (ServiceOmission), insignal2  in (ServiceOmission))
      ->  outsignal=ServiceOmission;
};
end;

thread interface compute is
insignal : in feature;
outsignal : out feature;

computer3: flow insignal ->  outsignal;
@EM{
    es: state threestate;
    e1: generator ;
    r1: transition e1 -> es=s1 when es = s0;
    computer2: token insignal in (ServiceOmission) ->  outsignal=ServiceOmission when es in (s1);
    r3: token insignal in (ServiceOmission) -> sink when es in (s0);
};
end;
```

# EMV2 Equivalent Example - 2

```
system conntop.i is
    sense1: abstract sensor.i;
    sense2: abstract sensor.i;
    processing: process controlProcess.impl;
    actuate: abstract actuator.i;
    hw : system hardwareplatform.impl;
    sensetocontrol1: connection sense1.outp -> processing.insignal1;
    sensetocontrol2: connection sense2.outp -> processing.insignal2;
    controltoactuate: connection processing.outaction -> actuate.inp;
    effectprop: connection actuate.effect -> effect;
@EM{
    effect2:token actuate.actt2.effect in (ServiceOmission) ->  effect=ServiceOmission;
};
end;
```

# Implementation Prototype

Instance model with behavior rule and generator instantiation

JGraphT based graph representation superimposed on instance model for processing

- Basic component interaction
  - Sufficient for property based token source/sink analysis and default any to all propagation
- Feature specific component interaction
  - Takes into account behavior flow logic
- Behavior rule specific component interaction
  - Takes into account token condition logic

Forward and backward token trace representation

Backward trace representation optimization transformation into fault tree, minimal cut set