AADL 3
Type System and Expression Language

Lutz Wrage

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

Type System Unification

Unification of type systems and expression languages (Peter, Lutz*, Alexey, Brian, Serban)

- Data Components
- Property Types
- Classifiers
- Annexes
  - Resolute, AGREE
  - Data Modeling
  - EMV2
  - BA, BLESS
- ReqSpec
- Scripting languages (Python)

Current Composite Types

AADL 2.2

Property types

- Range of
- List of
- Record

Data implementations

No operations available except

- List append (+=>)
- Boolean operations

Property expressions provide syntax for literals only

ReqSpec adds expressions, uses basic type inference

Current Usages of Types

Application data that occurs in the modeled system

- Data subcomponents
  - Shared data
  - Local variables in threads and subprograms
- Data communicated via data and event data ports

Information about the modeled system and individual components

- Properties

Mixture of models and properties

- Component classifiers and model elements as property values
  - Bindings
  - Specify constraints, e.g., `Required_Virtual_Bus_Class`

Additions in annexes

- Resolute: sets
- EMV2: error types and type sets, error types can have properties
- BLESS

Type System and Expression Language Goals

Provide types for

- Properties
- Features, e.g., data ports
- Data components
- Error types(?)

Support

- Specification of dependencies / constraints between properties
- Selecting model elements in configurations: Queries
- Structural analysis of instance models
  - Similar to Resolute
- Requirement specification
  - Similar to ReqSpec

Do we need structural analysis / constraints for declarative models?

Type System Unification Approach

Base types

- Integer, Real, Boolean, String
- Enumeration, Unit
- Category (`thread`, `processor`, etc.), Classifier, Model Element
- Range of Numeric (`Compute_Execution_Time => 10ms .. 15ms`)

Composite types

- List (ordered sequence of arbitrary length): `list of int`
- Set (unique elements): `set of classifier`
- Record (named fields) / Union (named alternatives)
- Tuples (unnamed fields)
  - Convenient for multiple return values from a function
- Map: `map mode -> Time`
  - Modal and binding specific property values in AADL 2.2 are (almost) maps
  - Error type specific property values
- Arrays: `array of int(10)`
- Bag (?)
- ~~Graph~~

Type System Unification Approach

Properties on types

Useful for code generation and analyses that looks at data size (in memory or on a bus)

- Information about representation
  `int {data_size => 16bit}`
- Range of valid values
  `int {range => 10 .. 20}`
- Size of a fixed size list (if we don't have arrays)
  `list of int {size => 3}`

Properties are ignored for type checking purposes

User Defined Types

Users can create named types

- `type byte: int { range => 0 .. 255 }`
- `type otherByte: byte { data_size => 8bit }`

- ```
  type sensed: record (
    value: int,
    timestamp: int
  )
  ```

```
type sensed2: record (
  value: int,
  timestamp: int
)
```

Is a type name just a shorthand, or is it a new type?

- Structural equality is easily implemented, but we may want the same type **name** on connected ports
- Fully “opaque” types would complicate the expression language, i.e., how would we know that we can add 2 bytes?
```

Numeric Ranges

Subsets of numeric types (or enumerations?)

- Range constrained Numeric
  e.g., `int [100 .. 120]`
- Could be considered special syntax for a property on a type
  e.g., `int {range => 100 .. 200}`

Subset constraints are difficult to maintain for expressions

- Simple assignments are easy to check
- If x is an integer [100 .. 120]
- 2 * x results in integer [200 .. 240]
- sqrt(integer[100 .. 120]) results in (not quite) real[10.0 .. 10.95]

Type checking should ignore range constraints, maybe except for simple assignments

Expression Language: Literals

Numbers, strings, boolean true/false as in AADL 2

- Automatic conversion from integer literal to real value

Range literals

- AADL2: `2 .. 3` or interval notation `[2, 3]`

Enumeration and unit literals

- Qualified name: <package>.<enum type>.<enum literal>
  e.g., `myenums.signaltype.RED`
- Need to import enumeration and unit literals in order to use their simple names

Collections

- To mirror declaration syntax
- **`list`** `(1,2,3)` is a **`list of int`**
- **`record`** `(intfield = 1, boolfield = true)` is a
  `record` `(intfield:` **`int`**`, boolfield:` **`bool`**`)`

Expression Language: Operations 1

Boolean

- and, or, not, …

Numeric values

- +, -, *, /, div, mod

Ranges

- Union, intersection, contains

Enumerations

- Consider them ordered, comparison operations

Units

- Get conversion factor, conversions

Strings, List

- append, substring, …

Records

- Access a field value

Union

- Access field depending on variant tag

Expression Language: Operations 2

Set

- union, intersection, contains

Generic collection operations

- forall, exists, filter, fold
- Look for inspiration in existing collection library and copy

Classifiers

- Extends, get extended, get all extending, …
- → methods defined in the AADL meta-model

Named elements

- Get name, get classifier, get all subcomponents, …
- → methods defined in the AADL meta-model

Variables

Need to be able to name results of expressions

- `val x = 2 * 5`

Variables or unmodifiable values?

- For constraints and structural analysis unmodifiable named values should be sufficient
- Variables require additional language constructs (loops) that can be avoided if only named values are allowed

Add vals in block expressions

- ```
  {
      val x = 2;
      x + 1
  }
  ```
  ```

Function Definitions

Reusable expressions => Functions

Proposed syntax

```
• def double(x: int): int = 2 * x
• def triple(x: int): int = {
      val d = double(x);
      x * d
  }
• def factorial(x: int): int = {
      def f(x: int, a: int): int =
            if x <= 1 then 1 else f(x-1, x * a);
      f(x, 1)
  }
```

Prototype Implementation

Expression Annex for AADL2

Implemented

- Most types
- Some type checking
- Subset of expressions
- Initial expression evaluation

- No units yet

Type Extension

Type extension

- Exists for classifiers to add subcomponents, properties, …
- Records
  - Add fields
- Unions:
  - Add fields to one or more variants(?)
  - Add variants
- Add properties to any type
  - `byte` is a subtype of `integer`
  - Not problematic as properties are ignored for type checking
- Assignment compatibility and type inference
  - `list of byte` is subtype of `list of integer`
  - Should be possible to define in a sound manner

Should there be configurations for types?

Measurement Units

Represent a (physical) quantity as a number with a dimension

- Length, Time, Mass, Force

Dimension has associated measurement units

- Length – **m**eter (SI base unit)
- Time – **s**econd (SI base unit)
- Mass – **k**ilo**g**ram (SI base unit)
- Force – **N**ewton (Derived: $1\,N = 1\,\frac{kg \cdot m}{s^2}$)

Different unit systems

- SI vs. Imperial
- Non-physical quantities, e.g., bit, byte
- Other: minute, day, year; rpm, angle, …

Users must be able to define new units

Unit Definition 1

Defining dimensions and corresponding measurement units

- Dimension as variation of enumeration types
  - `type LengthU: unit (cm, m = 100 * cm, …)`
  - `type TimeU: unit (s, ms = s / 1000, …)`
  - `type USLengthU: unit (in, ft = 12 * in, …)`
- Similar to AADL2
- Similar to compound type declarations (records, lists, etc.)

Literals with units

  - `100 ms`
  - `12 [ms]`

Type declarations with units

- `type LengthType: real [LengthU]`
- `type LengthType: real unit LengthU`

Unit Definition 2

Property definition

- Value is a physical quantity
  - `**property** distance: real **unit** USLengthU`
  - `**property** distance: real [USLengthU]`

  - `distance => 2.5 [in]`

- Value is a unit, e.g., to document the unit of the data on a data port
  - `**property** dataUnit: LengthU`

  - `dataUnit => [m]`

Standard Metric Prefixes

Metric prefixes

- Base 10: **c**enti, **m**illi, micro **μ**, **d**eka, **k**ilo, **M**ega
- Binary: **Ki** (2^{10}), **Mi** (2^{20}), **Gi** (2^{30})
- These are case sensitive, one is a greek letter
- Not distinct from units: **m**eter vs. **m**illi

Convenient to use them with any unit without repeatedly defining the conversion factor.

Use syntax to separate metric prefix and unit name

- `1 [k’g], 12 [m’s],` 640 `[Ki’byte]`

Only with base units

- If ms is defines as derived (ms = s/1000) the `1 [k’ms]` should not be valid

Unit Expressions 1

Avoid units names such as `KBytesps` (as we have in AADL 2)

Allow expressions for derived units

- `[k’g * m / s^2]`

Unit expressions are written in [ ]

- `speed == 12 [m/s]`

Simple unit may be written with or without [ ]

- `latency == 10 m’s` or `latency == 10[m’s]`

Allow only multiplication, division, and exponentiation

Defining a derived unit type

- **`type`** `ForceU =` **`unit`** `(N = [k’g * m / s^2])`

Unit Expressions - 2

Convert between numbers and quantities

- `val x = 1` x is an integer
  `val y = (x + 1)[s]` y is an integer with a unit: 2s
  `val z = y in [ms]` z is an integer: 1000

Calculation with units

- `10 N / 2.5 k’g == 4.0 [m / s^2]`

Unit Definitions and Usage

Derived units with unit expressions

- **type** MassU: **unit** (g)
- **type** SpeedU: **unit** (LengthU / TimeU)
- **type** ForceU: **unit** (N = k’g * m / s^2, …)

Type declarations with units

- **type** SpeedT: real [SpeedU]
- **type** ForceT: real [ForceU]
- **type** OtherSpeedT: real [LengthU / TimeU]

Property definition

- **property** speedUnit: Speed
- speedUnit => [m/s]
- **property** force: ForceT
- speed => 2.5 [k’g * m / s^2]

Expressions and Classifiers

Add **val**s and **def**s to classifiers

Specify expressions that should be evaluated

```
system S.i
    -- subcomponents, etc
    prop => 1;
    val v = 1;
    def f(x: int): int = x;
    -- assertions or invariants
    assert test: #prop == f(v);
end S.i;
```

Definitions and assertions are inherited or can be configured in

For structural verification

- Add descriptive text to assertions (similar to Resolute claim functions)
- Analysis evaluates assertions (all, or just for a single component) on an instance model

Next Steps

Complete expression annex implementation

Work out details of type extension

Add types and expressions to AADL 3 prototype implementation

Draft document