Unified Type System

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 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 properties
  - 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

Current Composite Types

AADL 2.2

Property types

- `range of`
- `list of`
- `record`

Data implementations

- Very similar to records

No operations available except

- List append (+=>)
- Boolean operations

Property expressions provide syntax for literals

ReqSpec adds expressions, uses simple type inference

Goals

Goal of the type system

Common type system available for use as data types, property types, annex sublanguage types

Goal for today

Discuss the scope of the type system

What should be in AADL 3 and what is outside?

Type System Unification Approach

Goal: *Common type system available for use as data types, property types, annex sublanguage types*

Base types

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

Composite types

- Array (ordered sequence of fixed length)
- List (ordered sequence of arbitrary length)
- Record (named fields)
- Union (named alternatives)
- Tuples (unnamed fields)
- Set (unique elements)
- Map
  - Modal and binding specific property values in AADL 2.2 are (almost) maps
- Bag, Graph

Which should be built-in?

Properties on Types

AADL2 core language, data modeling annex, and code generation annex use data classifiers to associate properties to data types for a variety of purposes

Meta data for use by model analyses

- `integer {data_size => 16bit}`

Restrictions on valid values

- `integer {range => 100 .. 1000}`

Information for code generation

- `string {encoding => UTF-16}`

Could be used for measurement units(?)

- `integer {unit => Time}`

What should a type checker do w.r.t. properties?

- *Interpret some pre-declared properties?*
  - Allows checking of assignments: -1 is not valid for `integer {signed => false}`
  - Details may become complicated
- *Ignore properties completely?*
  - No checking of assignments
  - Properties are then not part of a value of the type so no need to track them
  - Need syntax for measurement units
- *Add properties to ports directly, not as part of the data type?*

User Defined Types

Users can create named types

- `type byte is integer { range => 0 .. 255 }`
- `type otherByte is byte { data_size => 16bit }`
- ```
  type sensed is record
    value: integer
    timestamp: integer
  end
  ```
  ```
  type sensed2 is record
    timestamp: integer
    value: integer
  end
  ```

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

- In OSATE we require that property types are identical for assigning one property to another
- AADL 2.2 standard:
  - property types must “be identical”, “match”
  - `Classifier_Matching_Rule` for port data types

Properties not part of type => otherByte, byte, integer are the same type

No structural comparison => sensed, sensed2 are different types

These user defined types correspond to data components in AADL2
  ```

Subtypes

Subsets of numeric types (or enumerations?)

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

Subset constraints are difficult to maintain for expressions

- Simple assignments are easy to check
- 2 * integer[100 .. 120] results in integer[200 .. 240]
- sin(integer[100 .. 120]) results in (not quite) real[-1.0 .. 1.0]

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

Type Hierarchy

Type extension

- Exists for classifiers (including data components in AADL 2)
- Records
  - Add fields
- Unions:
  - Add fields to one or more variants
  - Add variants
- Add properties to any type
  - `byte` is subtype of `integer`
  - Then: `list of byte` is subtype of `list of integer`
- Change property values?
- “refinement”?
- Should there be a complete type hierarchy with something like `Object` as the root?

Do we really need type extension for data types?

Expression Language: Literals

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

- Automatic conversion from integer literal to real value

Range literals

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 them

Expression Language: Operations 1

Boolean

- And, or, not, …

Numeric values

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

Ranges

- Union, intersection, contains

Enumerations

- Assign a numeric value to enumeration elements?
- Consider them ordered?

Units

- Get conversion factor

Strings, List

- Append, substring, …

Records

- Extract a field

Union

- Access field depending on variant tag

Expression Language: Operations 2

Classifiers

- Extends, get extended, get all extending

Named elements

- Get name, get classifier

Set

- Union, intersection, contains

Generic collection operations

- Forall, exists, filter

Does it make sense to define our own set of collections and operations on them or should we just borrow from an existing standard?

- OCL?, a programming language?

Expression Language: Operations 3

Where should AADL end?

- Types
- Values: properties, ports, constants
- Literals
- Expressions
- Functions
- Recursive functions (to process lists)
  - Now we have a programming language!

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
- Other: minute, day, year; rpm, angle, …

Users must be able to define new units

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

Should metric prefixes be part of the unit literal as in AADL2 or separate entities?

Unit Definitions and Usage

Defining dimensions and units

- `type LengthU is <cm, m = 100 * cm, …>`
- `type TimeU is <s, ms = s / 1000, …>`
- `type USLengthU is <in, ft = 12 * in, …>`

Type declarations with units

- `type LengthType is real <LengthU>`

Property definition

- Value is a unit
  - `property lengthUnit: LengthU`
  - `lengthUnit => <m>`
- Value is a physical quantity
  - `property distance: real <USLengthU>`
  - `speed => 2.5 <in>`

Should one be able to convert between different unit systems?

E.g. USLength ⇔ SI Length, TempF ⇔ TempC

Unit Definitions and Usage

Derived units with unit expressions

- **type** Mass **is** <kg>
- **type** Speed **is** <Length / Time>
- **type** Force **is** <N = kg * m / s^2, …>

Type declarations with units

- **type** SpeedT **is** real <Speed>
- **type** ForceT **is** real <Force>
- **type** OtherSpeedT **is** real <Length / Time>

Property definition

- **property** speedUnit: Speed
- speedUnit => <m/s>
- **property** force: ForceT
- speed => 2.5 <kg * m / s^2>

Should unit expressions be part of AADL3?

Data Subcomponents 1

How to model shared data without data components

- `sharedData: data MyRecord`

Types alone are not sufficient because they don't have access features!

Option 1: Interface extends type

- ```
  interface I extends MyRecord is
    …
  end
  ```
- `sharedData: interface I`
  ```

Data Subcomponents 2

Option 2: Generic component + data type as property

- ```
  component C is
    …
  end
  ```
- ```
  component implementation C.i is
    data_type => MyRecord
  end
  ```
- `sharedData: component C.i`

Issues

- No enforcement of consistent extension between implementation and data_type property
- “Magic property”

Option 3: Interface + data type as property

- ```
  interface I is
    …
  end
  ```
- `sharedData: interface I {data_type => MyRecord}`

Issues

- Same extension consistency issue if property is included in interface

```
interface I is
 data_type => MyRecord
end
```
```

Type System Usage

Properties

- Property definitions reference types
  `property temp: TemperatureT`

Port types

- Associate a data type directly with a port
  `p: in port TemperatureT`

Data representation

- Data modeling annex
- `Base_Type` property references a data classifier
- If we still need that, the type of `Base_Type` must be “type” or “data”

Representation of Transferred Data

Example

- `type BodyTemp is integer <TemperatureUnits>`
  `p1: out port BodyTemperature;`

Is unit included in transferred data or is a unit assumed?

- `p1: out port integer {unit => <degC>}`

Non-zero reference point for transferred value

Transfer representation may be different from in memory representation

Alternatives:

- Protocol specification
  - As virtual bus
  - Mapping into bit representation (see 429 protocol example in SAVI demo)

Representation of Types

Example

- BodyTemperature: **type** integer [30..50 C] **units** TemperatureUnits;
- P1: **out port** BodyTemperature;

Digital representation

- Base_Type property in Data_Model
- Associated with type or with port

Physical representation

- Dynamic behavior
- Specified as part of type or specific to each use site
  - Associated with feature