Multi-Level Specification Language (MLSL)

Syntax definition

1. Introduction

This document briefly describes the instruction set of MLSL, a textual language used to specify the structure and semantics of multi-level constructs in the Multi-Level Modeling Playground (MLMP). The document focuses mostly on the syntax of the instruction set of the language. Instructions are presented in EBNF (Extended Backus-Naur Form)[1], focusing on readability and comprehensibility over preciseness and the accurate and full description of the grammar. Essentially, this document is a brief user manual for MLSL, also showcasing simple examples.

Note that MLMP is an ongoing research project that may improve based on user (researcher) feedback. Therefore, MLSL is also subject to change and this document could also be updated in time. We welcome any feedback on MLMP and MLSL; if you have any, feel free to contact us at *dmla@aut.bme.hu*.

Finally, for further examples, see the MLMP repository[2] (the link can also be found on the DMLA website) for the mapping of several existing multi-level constructs in the literature. The website and repository are also subject to future changes. Feedback on mapped constructs is also much appreciated!

[1] https://en.wikipedia.org/wiki/Extended_Backus-Naur_form
[2] https://github.com/bmeaut/MLMP

2. Module definition

2.1.Structural mapping

Structural entity definition

```
structural entity = name, “:”, structural meta type, “{“,
 {structural attribute}, “}” ;

name = identifier ;
structural meta type = qualifier ;
identifier = letter, {(letter | digit | “_”)} ;
letter = “a” | … | “z” | “A” | … “Z” ;
digit = “0” | … “9” ;

number = [“-“], {digit} ;
qualifier = identifier, {“.”, identifier} ;
```

Note that in later production rules, some of the previous rules (like name or identifier) will be re-used and refer to the ones defined above. The number rule is shown here only for this reason.

Structural attribute definition

```
structural attribute = name, “:”, structural meta type, [array] “;” ;
array = “[]” ;
```

The structural meta type can either refer to an existing structural entity or attribute, or to a member of a pre-defined set from which primitive DMLA types will be generated: *entity, slot, slot.value, slot.type, bool, float, int, string*.

Examples

```
ModelElement: entity {}

Node: ModelElement {
  fields: Field[];
  isAbstract: bool;
}
Field: slot {
  value: slot.value;
  type: slot.type;
}
```

2.2. Annotation definitions

```
annotation = [override], mutability, name, optionality, “in”, “{“,
range, “}”, “on”, targeted elements ;

override = “override” ;
mutability = “immutable” | “mutable” ;
optionality = [“?”] ;

range = interval | set of literals ;
interval = interval boundary, “..”, interval boundary ;
interval boundary = number | “*” ;
set of literals = name, {“,”, name} ;

targeted elements = qualifier, {“,”, qualifier} ;
```

Examples

```
mutable level in {0..*} on ModelElement, Field;
override mutable potency in {-1..*} on ModelElement, Field;
```

2.3.Validation rules

Omission rule and if statement definition

```
validation rule = omission rule | statement ;
statement = if statement | simple statement ;

omission rule = “omit”, rule identifier, {“,”, rule identifier}, “;” ;
rule identifier = “#”, identifier, “:” ;

if statement = “if”, “(“, condition, “)”, “{“, {statement}, “}”,
 “else”, if statement | “{“, statement, “}” ;
condition = expression ;
```

Other statement definitions

```
simple statement = expression, [where clause], “;” ;

where clause = “where”, (expression | in clause),
 {“and”, (expression | in clause)} ;
in clause = name, “in”, name ;

expression =    simple expression                              |
                “!”, expression                                |
                sum                                            |
                expression, arithmetic op, expression          |
                expression, logical op, expression             |
                expression, “and” | “or”, expression           |
                qualifier, “in”, closure                       |
                quantifier, expression                         |
                “(“, expression, “)” ;

simple expression = qualifier | number | “true” | “false” | ”null” ;
sum = “sum”, “(“, qualifier, “)” ;
arithmetic op = “+” | “-“ | “*” | “/” ;
logical op = “=” | “!=“ | “<” | “>”| “<=”| “>=” ;

closure = “closure”, “(“, qualifier, “)”, closure body ;
closure body = “{“, statement, “}” ;

quantifier = “forall” | “exists”, variable definition, “:” ;
variable definition = name, “as”, variable type ;
variable type = identifier ;
```

Examples

```
potency <= level;
Field.level = Node.level where Field in Node.fields;
ModelElement.level = ModelElement.meta.level - 1;

if (ModelElement.meta.potency != -1) {
    ModelElement.potency = ModelElement.meta.potency - 1;
}

#B7: forall inh as Inheritance:
  forall f2 as Field:
   exists f1 as Field:
    f2.name = f1.name and f2.type = f1.type and
    f2.value = f1.value
 where f1 in inh.source.fields and f2.potency > 0 and
  f2 in inh.target.fields;

#B15: ! exists inh as Inheritance: inh.source in closure(inh.target) {
    Inheritance.target where Inheritance.source = inh.target;
};
```

3. Domain definition

3.1. Structural mapping

Structural entity definition

```
structural entity = name, “:”, structural meta type, “{“,
 {structural attribute}, “}” ;
structural meta type = identifier ;
```

Note that the production rules defined here (like structural entity) may share the same name with their counterparts in the module definition, there are differences in the definition. Production rules that are not re-defined here (like name or digit) refer to their old definitions.

Structural attribute definition

```
structural attribute = simple attribute | complex attribute ;

simple attribute = [metric marker], name, “=”, simple value, “;” ;
metric marker = “@” ;
simple value = qualifier | number | “null” | “true” | “false” |
 string | “*” ;
number = [“-“], {digit}, [“.”, {digit}] ;
string = '"' , {all_characters - ('"', new_line)}, '"' ;

complex attribute = name, “{“, structural entity, “}” ;
```

Rule all_characters represents every character, but its definition is omitted due to reasons of compactness. Rule new_line represents new line characters similarly.

Examples

```
Configuration: Node {
  @potency = 3;
  @level = 3;
  isAbstract = false;

  fields {
    salesPrice: Field {
        @potency = 2;
        @level = 3;
        @nature = simple;
        type = int;
    }

    name: Field {
      @potency = 3;
      @level = 3;
      @nature = dual;
      type = string;
    }
  }
}

hasFrontWheel: hasWheels {
  @potency = 0;
  @level = 0;
  source = Bike134123;
  target = FrontWheel;

  sourceMin = 1;
  sourceMax = 1;
  targetMin = 1;
  targetMax = 1;
}
```