DMLA Elements

A. The Core

In DMLA, the model is represented as a Labeled Directed Graph. Each element of the model such as nodes, edges or even attributes can have arbitrary labels. These labels are used either to hold data (e.g. concrete literal value of an attribute) or to express relations (e.g. containment) between the elements. Because attributes may have complex structure, we represent them as hierarchical trees. Also, for the sake of simplicity, we will use a dual field notation for labelling of Name/Value pairs, that is, a label with the name N of the model element X is referred to as X_N.

1) Labels and universes

In DMLA, we defined the following labels: (i) X_{ID}: globally unique ID of model element; (ii) X_{Meta}: ID of the meta-model definition; (iii) X_{Values}: values of the model element; (iv) $X_{Attributes}$: ordered set of contained attributes.

Definition – The superuniverse |A| of a state A of the DMLA consists of the following universes: (i) U_{Bool} containing logical values {*true/false*}; (ii) U_{Number} containing rational numbers and a special symbol ∞ representing infinity; (iii) U_{String} containing character sequences of finite length; (iv) U_{ID} containing all possible entity IDs; (v) U_{Basic} containing elements from $\{U_{Bool} \cup U_{Number} \cup U_{String} \cup U_{ID}\}$. Additionally, all universes also contain a special element, undef, which refers to an undefined value.

The labels of the entities take their values from the following universes: (i) X_{ID}: U_{ID}, (ii) X_{Meta}: U_{ID}, (iii) X_{Values}: $U_{Basic}[]$, (vi) X_{Attrib}: $U_{ID}[]$. The label Values is an ordered list of primitive values, while Attrib is a set of IDs, which refer to other entities.

2) Functions

In ASMs, functions are used to rule how one can change the states. In DMLA, we rely on shared and derived functions. The current attribute configuration of a model element is represented via shared functions. The values of these functions can be modified either by the algebra itself, or by the environment of the algebra (e.g. by the user). Derived functions represent calculations which cannot change the model; they are only used to obtain and to restructure existing information. The vocabulary $\sum$ of DMLA is assumed to contain the following characteristic functions: (i) Meta(U_{ID}): U_{ID}, (ii) Attrib(U_{ID}, U_{Number}): U_{ID}, (iii) Value(U_{ID}, U_{Number}): U_{Basic}. The functions are used to access the values stored in the corresponding labels. These functions are not only able to query the requested information, but they can also update it. For example, one can update the meta definition of an entity by simply assigning a value to the Meta function (although the new relation may be invalid based on the instantiation rules). Moreover, there are two other derived functions: (i) Contains(U_{ID}, U_{ID}): U_{Bool} and (ii) DeriveFrom(U_{ID}, U_{ID}): U_{Bool}, which check containment and instantiation (transitive) relations, respectively.

B. The Bootstrap

The ASM functions define the basic structure of the algebra and also allow to query and change the model. However, relying only on these constructs, it would be hard to use the algebra in practical modeling scenarios due to the lack of basic built-in data constructs. For example, entities are required to represent basic types; otherwise one cannot use label Meta when it refers to a string since the label is supposed to take its value from U_{ID} and not from U_{String}. Hence, one must be able to define those Base constructs somewhere inside or outside the core algebra. Obviously, there may be more than one such "correct" solution to define this initial set of information. We introduced the concept of the bootstrap which is a flexible and swappable layer for defining the basic modeling elements. The particular bootstrap we will present in this paper is a generic one that can be used as root of any domain-specific bootstraps.

Semantically, modeling entities of the bootstrap (Fig. 1) can be categorized into four groups: (i) basic types providing a basic structure for modeling, (ii) built-in types representing the primitive types available in DMLA, (iii) entities used in describing operations, and (iv) validation related entities.

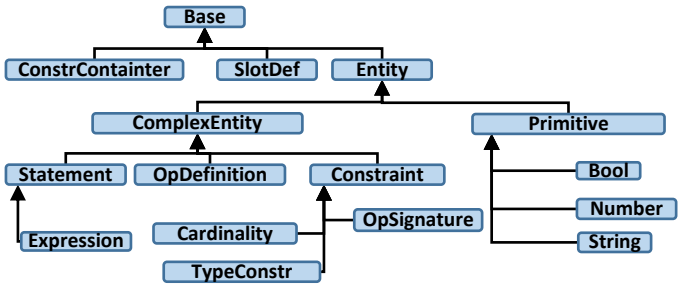

Fig. 1 Main elements of the bootstrap

1) Basic entities

Basic entities are basic building blocks, the hierarchy of modeling entities rely on these elements. Here, we present only a conceptual overview of the basic entities, mostly focusing on their role rather than getting on their exact, precise structure.

The *Base* entity is at the root of the whole DMLA modeling entity set, thus all other model entities are instantiated from it (directly or indirectly). *Base* defines that modeling entities can have slots (defined by *SlotDefs*) and *ConstraintContainers*. Slots represent substitutable properties, in syntactically similar manner to class members in OO languages. *ConstraintContainers* (and the contained *Constraints*) are used to customize instantiation validation formulae. Moreover, *Base* has two other slots, reserved for validation formulae that formalize the basic principles of the instantiation validation as explained later.

The *SlotDef* entity is a direct instantiation of *Base*. It is used to define slots. Slots can contain *ConstraintContainers*, which grants them the ability to attach constraints to any containment relation defined by the slot. Moreover, *SlotDef* overrides validation slots derived from *Base*.

The *Entity* entity is another direct instance of *Base*. *Entity* is used as the common meta of all primitive and user-defined types. *Entity* has two instances: *Primitive* (for primitive types) and *ComplexEntity* (for custom types).

2) Built-in types

The core entities needed to represent the universes of ASM in the bootstrap are: *Bool*, *Number* and *String*. All these types refer to sets of values in the corresponding universe. For example, we create entity *Bool* so that it could be used to represent Boolean type expressions. Built-in types are relied on when a slot is filled by a concrete value and that value is not a reference to another model entity, however it is a primitive, atomic value. All built-in types are instances of *Primitive*.

3) Operations

Operations are defined by AST representation consisting of model entities. Hence, the bootstrap must contain several such entities, for example, there is an *If* entity representing the semantics of the usual *if* statement, and similarly there is a *While* entity, which stands for ordinary *while* loop semantics. The most important such AST related entity is the *OperationDefinition*, which is used to define an operation. It has slots for a return type, a context, and certain number of parameters, including a special slot, called Body. The Body describes the logic of the operation ant it is a *Statement*. *Statement* entities play similar roles as statements in state-of-the-art programming languages, e.g. literals, relational expressions, conditional statements, blocks, iterations etc.

Besides defining the operations, one also needs some mechanism to invoke them. The *OperationCall* entity provides this functionality. It has slots for a context (i.e. "this"), an arbitrary number of parameters and an *OpHandle* entity which refers to an operation definition. All child entities of operation calls are *Expression* entities, thus, they can contain a simple, or a complex expression to be evaluated.

It is often useful to specify the signature of an operation. Without this, we would have typeless function pointers, which are hard to use. In DMLA, operations may have a special constraint, *OperationSignature* describing their signature, i.e. the type of its parameters and result.

4) Validation

In DMLA 2.1, the validation logic is transformed from a universal set of rules to modularized and explicit collection of sub-formulae. The basic mechanism of DMLA 2.0 universal validation logic relied on the selection of two type specific formulae based on the meta-hierarchy of the element to be validated. The two types of formulae are referred to as alpha and beta. The alpha type formulae have been constructed to validate an entity against one of its instances, by simply checking if the "is-a" relation between the two elements can be verified. In contrast, the beta type is an in-context check: it is mainly needed in case an entity has to be validated against multiple related entities. For example, cardinality-like constraints can be evaluated by beta formulae due to the underlying one-to-many relation. Moreover, entities can copy or extend the validation logic of their meta entity, which grants a high level of flexibility and expressiveness. Hence, the validation approach of DMLA 2.0 seemed to be flexible enough at first sight since the validation logic can be extended gradually and according to the needs of the particular bootstrap entities. However, we also quickly realized that, on the other hand, bootstrap-dependent logic had to be provided on ASM level, which significantly weakened the aimed flexibility of the original bootstrap design. In DMLA 2.1, the integration of operations ASTs into the bootstrap allowed the bootstrap to contain executable logic. This design made it possible to migrate the ASM based validation formulae into the bootstrap, turning them into a self-contained part of the DMLA infrastructure.

The type-specific alpha and beta type formulae were moved into the *Base* entity. *Base* fully incorporates the generic instantiation validation approach of DMLA, while the instances of *Base* can specialize the standard validation formulae, changing the behavior of the entities in their sub-branches of the meta-tree. In other words, new entities within the model may provide their own specialized definition of valid instantiation, provided they do not contradict the standard validation rules imposed by *Base*.

Similar to the validation rules, the constraints have also been modularized in DMLA 2.1. The basic idea was that many aspects of the validation can be also generalized instead of them being repeatedly encoded. Thus, we have defined a generic *Constraint* entity, which contains two new operations: constraint-alpha and constraint-beta. These operations are able to extend the alpha and beta formulae of the container of the *Constraint* instance. Also, we have removed the type and cardinality slots from the *SlotDef* entity, and have added *ConstraintContainer* to *Base* containing *Constraint* instances. The validation formula of *Base* now also calls these new operations of all *Constraint* instances contained.

Finally, we needed to address the problem of the life-cycle of the *Constraint* entities. In DMLA 2.0, the *SlotDef*-specific formulae were responsible to ensure that the cardinality and type constraint contained by *SlotDef* were copied (kept as they were), specialized (instantiated with stricter bounds) or discarded at the same time, which resulted in sealing the *SlotDef* instance, meaning it cannot be instantiated any further. When generalizing the constraint concept, the *SlotDef* lost its ability to "tend" for its contained *ConstraintContainer and Constraint* instances the same way. To achieve the same consistent rules, but also to embrace the flexibility introduced with the *Constraint* entity, two new formulae have been defined in the *Constraint* entity: the *ConstraintLifeCycleAlpha* and *ConstraintLifeCycleBeta*. When *ConstraintContainer* instances are being validated, the *ConstraintContainer* invokes the *ConstraintLifeCycle* formulae of the contained *Constraint* instance. This enables the *Constraint* instances to define their customized life-cycle logic.

In summary, in DMLA 2.1, the validation of the bootstrap is based on three pairs of formulae: the alpha and the beta type validation formulae, which are applied to every entity of the bootstrap; the *ConstraintAlpha* and the *ConstraintBeta* formulae, which are extensions of the container entity's alpha and beta formulae; and finally the *ConstraintLifeCycleAlpha* and the *ConstraintLifeCycleBeta* formulae, which manage and validate the life-cycle of the *Constraint* instances.