International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:11, No:1, 2017

Multi-Level Meta-Modeling for Enabling Dynamic Subtyping for Industrial Automation

Zoltan Theisz, Gergely Mezei

Abstract**—**Modern industrial automation relies on service oriented concepts of Internet of Things (IoT) device modeling in order to provide a flexible and extendable environment for service meta-repository. However, state-of-the-art meta-modeling techniques prefer design-time modeling, which results in a heavy usage of class sometimes unnecessary static subtyping. Although this approach benefits from clear-cut object-oriented design principles, it also seals the model repository for further dynamic extensions. In this paper, a dynamic multi-level modeling approach is introduced that enables dynamic subtyping through a more relaxed partial instantiation mechanism. The approach is demonstrated on a simple sensor network example.

Keywords**—**Meta-modeling, dynamic subtyping, DMLA, industrial automation, arrowhead.

I. Introduction

THE meta-model based industrial automation frameworks could be the next wave of solutions to meet the challenges set by Industry 4.0. The open IoT infrastructure requires a fully service oriented conceptualization of industrial automation. Although the service layer interface hides the internal data structure of concrete embedded devices, data modeling still remains an important part of the final solution. The services must be registered in a flexible repository and they shall be available for look-ups. Therefore, the necessary modeling efforts must go well beyond the current static nature of state-of-the-art model building. Clearly, object-oriented programming or meta-model facilitated subtyping supports only design time modeling. Although traditional industrial automation solutions could be easily satisfied by a fixed categorization of embedded devices, the newer wave of SOA-based automation frameworks openly embrace System of Systems (SoS) design concepts of composability. Hence, corresponding modeling solutions must be researched and made available. A straight-forward candidate solution could be dynamic subtyping, which provides both the benefits of dynamic introduction of new types and the consistency of already defined types within the (sub-)systems.

In this paper, Section II introduces the Arrowhead framework, then Section III explains dynamic subtyping. Next, in Section IV, a dynamic multi-level meta-modeling approach is introduced in detail. Then, Section V shows how the approach can be applied to come up with a viable implementation for dynamic subtyping, which is also demonstrated through a simple example in Section VI. Finally, Section VII concludes the paper.

Zoltan Theisz is with Evopro Innovation, Budapest, Hungary (e-mail: zoltan.theisz@evopro.hu)

Gergely Mezei is with the Department of Automation and Applied Informatics at the Faculty of Electrical Engineering and Informatics of Budapest University of Technology and Economics, Budapest, Hungary (e-mail: gmezei@aut.bme.hu).

II. The Arrowhead Framework

The Arrowhead framework [1] aims to become one of the best technology answers currently available to the general challenges raised by the Industry 4.0 revolution in industrial automation. The framework is based on the concept of the IoT, it clearly follows the principles of the SoS, and it has been implemented along the well-known guidelines laid out by Service Oriented Architecture (SOA). All these individual facets have been united to facilitate collaborative industrial automation of networked embedded smart devices. The concept of a service within the framework is defined as a piece of information that can be exchanged between or among participating communication systems. Since the final goal of the framework is to provide SoS features, it must also support a great variety of different sensors and activators, or any arbitrarily combinations thereof, through unique and well defined set of registered interfaces. The services are reusable and can be provided via various systems by explicitly declaring them via communication profiles each possessing the following three main characteristics: transfer protocol, security mechanism, and data format. Within the framework, the concept of a system is defined as a composable entity (according to the well-known principles of SoS) that can produce and consume services. The system that is producing a service is labeled as the Service Provider, while the system that is consuming a service is referred to as the Service Consumer. Also, a particular Arrowhead system may play the role of a provider of one or more services or a consumer of some other services, or even both at the same time.

Core Arrowhead mechanisms make use of three mandatory framework-wide services: Service Registry, Orchestrator and Authorization. These three basic services combined with the above definition of components within the system(s) establishes an SoS adapted version of SOA principles in order to enable a new wave of industrial automation facilitated by smart embedded devices. As can be seen in Fig. 1, firstly, the Service Registry stores all meta-data of any registered services and provider systems. Secondly, Management sets up and orchestrates the communication channels in order to combine the services into application systems, eventually also incorporating their reconfigurability. Finally, Authorization ensures that only properly authorized systems may provide and/or consume services within the framework.

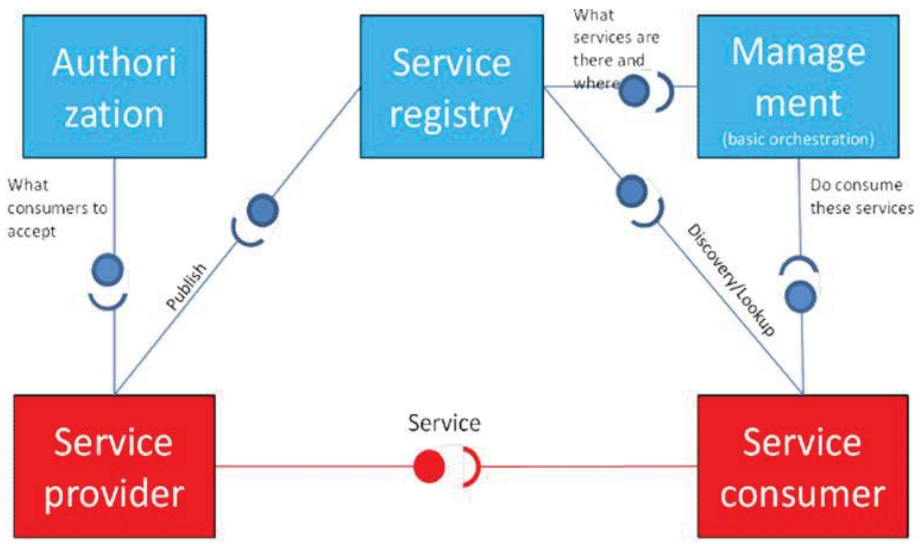

Fig. 1 Arrowhead's interpretation of SoS adapted SOA

III. Dynamic Subtyping

A. Need for Dynamic Subtyping

The Service Registry is the central modeling repository of the Arrowhead framework: it ties together both design-time and run-time aspects of service models covering the full life cycle of any particular service an industrial automation system may consist of. Hence, it is of paramount importance that it keeps on remaining both extendable and flexibly constrained. Thus, only proper services such as new compatible sensor instances are allowed to be inserted in or deleted from. On one hand, proper subtyping shall be needed to constrain the meta-data structure of the registered instances in the repository. On the other hand, it must be practical in the sense that new types may be defined dynamically by step-by-step addition of new or removal of already existing structural items in the defined meta-data structure. Moreover, sometimes particular service meta-data instances may be reused as templates for subtypes that extend those concrete values with further data structures. Hence, although legacy meta-model based subtyping is an effective workhorse for any object-oriented repository implementation, it may become very difficult to provide an extendable and type-correct internal mechanism for its opening towards the needs of Arrowhead compatible SoS based SOA.

B. Multi-Level Instantiation

Classical object-oriented modeling principles effectively restrict the number of meta-modeling levels to two: one of them being the class level where the data structure is defined, and the other one being the object level where those instances are stored which comply to the data structures. This restriction ultimately means that subtyping must be a "vertical" relation between the entities at the class level, while instantiation connects the class level entities to their corresponding instances at the object level. An unfortunate consequence of this strict isolation is that the class level gets frozen in design-time and cannot be further extended by new entities in run-time. However, in a multi-level meta-modeling architecture instantiation are allowed to take place across multiple levels. These techniques usually distinguish between two options: shallow instantiation means that the structural information is defined on the nth modeling level and it is used on the immediate instantiation (n+1)th level, while deep instantiation allows defining the information on the nth modeling level and use it on the (n+x)th ($x > 0$) modeling level [2]. Object-oriented instantiation principles support only shallow instantiation, which results in a rigid and unnecessary separation between class level and object level. This is the reason behind the fact that subtyping and instantiation are split and cannot be directly related to each other, one being "horizontal" and the other "vertical". Hence, legacy subtyping is static and instantiation is dynamic in nature, respectively. Therefore, dynamic subtyping should combine "horizontal" and "vertical" aspects by providing a means to be able to introduce new attributes and operations to the existing model entities in a precisely controlled manner. In a sense, the new dynamic subtyping concept will become a "horizontal" instantiation, in other words, it is carried out through the insertion of an extra meta-level.

IV. Dynamic Multi-Level Algebra

Dynamic Multi-Layer Algebra (DMLA) [3]-[5] is the theoretical framework facilitating the introduction of dynamic subtyping. It consists of three constituent parts: the first one defines the modeling structure and also establishes the ASM [6] functions declared on this structure. The second part creates the initial set of modeling constructs, the so-called minimal set of built-in entities which are needed to bootstrap any further modeling. Finally, the third part validates and also safeguards the correctness of the instantiation mechanism.

A. Modeling Structure

DMLA represents the model as a Labeled Directed Graph (LDG), where each node and edge can have labels attached. Those labels define the attributes of the model elements of the LDG. For simplicity, we apply dual field notation in labeling for Name/Value pairs. The data representation of the model elements is defined by the below labels:

- X_{Name}: name of the model element
- X_{ID}: globally unique ID of the model elementv
- X_{Meta}: ID of the meta-model definition
- $X_{Cardinality}$: cardinality of the model element
- X_{Value}: value of the model element
- $X_{Attributes}$: list of attributes

Now, let us define the algebra over those labels. The superuniverse $\mathfrak{A}$ of a state $\mathfrak{A}$ of the Multi-Layer Algebra consists of the following universes:

- U_{Bool}: containing logical values {true/false}
- U_{Number}: containing rational numbers and a special symbol ∞ representing infinity
- U_{String}: containing character sequences of finite length
- U_{ID}: containing all the possible entity IDs
- U_{Basic}: containing elements from { U_{Bool} U U_{Number} U U_{String} U U_{ID} }

Additionally, all universes contain a special element, undef, which refers to an undefined value. The labels of the entities take their values from the following universes:

- X_{Name}: U_{String}
- X_{ID}: U_{ID}
- X_{Meta}: U_{ID}
- $X_{Cardinality}$: [U_{Number} , U_{Number}]
- X_{Value}: U_{Basic}
- $X_{Attributes}$: U_{ID} []

The label Attrib is an indexed list of IDs, which refers to other entities. Now, let us have a simple example:

SensorID=69, SensorMeta=42, SensorCardinality=[0, inf], SensorValue=undef, SensorAttrib=[]

The above definition describes a Sensor entity with its ID set to 69, and the ID of its meta-model being 42. From now on, the same semantics will be shown in a more compact representation (both tuples and lists by square brackets).

{“Sensor”, 69, 42, [0, inf], undef,[]}

B. ASM Functions

ASM functions are used to control how one can change the states within the ASM. DMLA relies on both shared and derived functions. Shared functions show the current attribute configuration of a model entity. DMLA allows the values of these functions to be modified either by the algebra or by its environment. Derived functions represent calculations which cannot change the model; they only obtain or restructure existing entity information. The vocabulary ∑ of DMLA consists of the following shared functions (for technical details see [5]):

- *Name(ID)*
- *Meta(ID)*
- *Card (ID)*
- *Attrib(ID,Idx)*
- *Value(ID)*
- *Contains(ID_1,ID_2)*
- *DeriveFrom(ID_1,ID_2)*

C. Bootstrap Mechanism

The ASM functions have defined the basic structure of the algebra and the means to query and change the model. However, some initial built-in constructs are also needed for any practical modeling. As a minimum, the bootstrap creates the basic types and the so-called principal entities. The basic types correspond to DMLA’s defining universes such as Basic, Bool, Number, String, and ID. Besides these, the first two types of like entities are also defined: Attribute and Base. They act as root entities of all other attributes and node meta-types, respectively. Furthermore, a third entity, AttribType, is also defined that restricts, by corresponding validation formulae, the value of the attributes within the instances. Namely, the label Value of an AttribType specifies which type may be used within the instance given by the referred attribute. However, AttribType and its Value field must only be filled in if the given attribute has been instantiated; otherwise, AttribType can be totally left out. Nevertheless, AttribType can only be applied for attributes.

The principal entities of DMLA are defined as follows:

```
{“Attribute”, IDAttribute, IDAttribute,
[0,inf], undef,
[
 {“Attributes”, IDAttributes, IDAttribute,
   [0, inf], undef,[]}
]}

{“Base”, IDBase, IDBase,
[0, inf], undef,
[
 {“Attributes”, IDAttributes, IDAttribute,
  [0, inf], undef,[]}
 {“Links”, IDLink, IDAttribute,
  [0, inf], undef,[]}
]}

{“AttribType”, IDAttribType,IDAttribute,
[0,1],undef,
[
 {“AType”, IDAType, IDAttribType,
  [0,1],IDID,[]}
]}

{“Node”, IDNode, IDBase,
[0, 0], undef,
[
 {“Attributes”, IDAttributes, IDAttribute,
  [0, inf], undef,[]}
]}
```

D. Validation Formulae

During the instantiation process, only those instances are allowed to be created that do not violate any constraints set by the meta-definitions. Thus, DMLA distinguishes between valid and invalid models, where validity checking is based on

formulae. Hence, it is assumed that whenever external actors change the state of the algebra, those formulae must be evaluated. This dynamic property enables DMLA to be applied both in design-time and run-time. The mathematical details of the formulae are published in [4].

Validation formulae take an Instance entity and a MetaType entity, and then check if the Instance entity is a valid instance of the MetaType entity. However, the formula φMeta takes only one parameter and validates if the given entity has enough valid instances according to the cardinality label.

- $\varphi_{IsInstantiated}(C, I)$: checks if *I* is instantiated in *C*
- $\varphi_{InstCounter}(C,a_1,a_2)$: checks if two attributes in the same container *C* are originated from the same meta definition
- $\varphi_{CardinalityCheck}(C, I)$: checks if an attribute *I* violates the cardinality constraints in the container *C*
- $\varphi_{Typecheck}(T, v)$: checks if a specific value *v* violates the type constraint *T*
- $\varphi_{AttribCheck}(I, a)$: checks if an attribute *a* is a valid instantiation of a meta attribute of the meta of container *I*
- $\varphi_{LabelCheck}(I, M)$: checks the correct usage of *Meta* label
- $\varphi_{EntityIns}(I, M)$: checks if at least one of the attributes is instantiated, or has its value set
- $\varphi_{AttribSrc}(I, M)$: checks if there is an attribute violating the cardinality constraint or not having a valid meta definition in the meta of the entity
- $\varphi_{ValueCheck}(I, M)$: checks the validity of the *AttribType* definitions
- $\varphi_{Link}(I, M)$: checks if instances of *Base* have the correct number of links
- $\varphi_{Meta}(M)$: checks the cardinality limits
- $\varphi_{IsValid}(I, M)$: checks the validity of the new ASM state is by combining $\varphi_{LabelCheck}(I, M)$, $\varphi_{AttribSrc}(I, M)$, $\varphi_{EntityIns}(I, M)$, $\varphi_{AttribType}(I, M)$ and $\varphi_{Link}(I, M)$

V. Dynamic Subtyping via DMLA

DMLA provides the means for dynamic subtyping by simply constraining the generic instantiation mechanism to a special subcase. Namely, dynamic subtyping is only allowed to modify the structure of the meta-entity, but no value substitution must take place there. More precisely, the original formula of φAttribCheck(I, a) used to be:

$\varphi_{AttribCheck}(I, a)$:
$\varphi_{IsValid}(a, Meta(a)) \vee$
$\exists j$: Attrib(Meta(I), j)=a $\vee$
{$\exists m$, k: m=Attrib(Meta(I), k) $\wedge$ Value(m)=Value(a) $\wedge$ Name(m) = Name(a) $\wedge$ Meta(m) = Meta(a) $\wedge$ ($\exists i$: Attrib(a,i) $\neq$ Attrib(m,i))}}

The formula has three parts which check if an attribute is:

- either a valid instantiation
- or a copy
- or a meta

of a meta attribute of the meta of container *I*. Therefore, if this formula is slightly modified it can easily grasp the essence of dynamic subtyping in the following way:

$\varphi_{AttribCheck'}(I, a)$:=
{$\exists m$, k: m=Attrib(Meta(I), k) $\wedge$ Value(m)=Value(a) $\wedge$ Name(m) = Name(a) $\wedge$ Meta(m) = Meta(a) $\wedge$ ($\exists i$: Attrib(a,i) $\neq$ Attrib(m,i))}}

In other words, dynamic subtyping only copies existing structures or introduces new structures if related cardinality constraints allow that. Hence, DMLA can be easily incorporate dynamic subtyping by restricting its original instantiation formula.

VI. Example

In order to showcase the subtyping and the instantiation mechanism of DMLA, let us examine a simplified Arrowhead network device scenario. In this scenario, an "abstract" SensorType service entity is defined that allows an arbitrary number of wireless communication channels for data collection and also provides a single IP address for network traffic.

```
{"SensorType", IDSensorType, IDNode,
 [0, inf], undef,
 [
{"WirelessChannel",IDWirelessChannel,IDAttribute,
   [0,inf],undef,
   [
    {"WCType",IDWCType,IDAttribType,
     [0,inf], IDString,[]}
   ]
  }
{"IPAddr",IDIPAddr,IDAttribute,
   [1,1],undef,
   [
    {"IPType",IDIPType,IDAttribType,
     [0,inf], IDString,[]}
   ]
  }
 ]}
```

This SensorType service entity is only a meta-type definition; therefore, a service must first instantiate it, for example, by assigning a particular IP address to its network interface, in order for it to be registered into the Service Registry.

```
{"MySensor", IDMySensor, IDSensorType,
 [0, inf], undef,
 [
{"WirelessChannel",IDWirelessChannel,IDAttribute,
   [0,inf],undef,
   [
    {"WCType",IDWCType,IDAttribType,
     [0,inf], IDString,[]}
   ]
  }
 {"IPAddress", IDIPAddress, IDIPAddr,
  [1,1], "192.168.0.1", []
 }
]}
```

In a classical subtyping implementation, the type of the service cannot change any longer, but if dynamic subtyping has been enabled, then

- either a new "concrete" type can be derived from SensorType
- or the MySensor instance can dynamically change its structure

by changing the cardinality of WirelessChannel. Let us assume that entity SpecificSensor must have at least five wireless channels and the maximum amount is eight. In the first case, the result looks like as follows:

```
{"SpecificSensor", IDSpecificSensor, IDSensorType,
[0, inf], undef,
[
{"WirelessChannel",IDWirelessChannel,IDAttribute,
  [5,8],undef,
  [
   {"WCType",IDWCType,IDAttribType,
    [0,inf], IDString,[]}
  ]
 }
{"IPAddr",IDIPAddr,IDAttribute,
  [1,1],undef,
  [
   {"IPType",IDIPType,IDAttribType,
    [0,inf], IDString,[]}
  ]
 }
]}
```

In the latter case, the definition is the following:

```
{"MySensor", IDMySensor, IDSensorType,
[0, inf], undef,
[
{"WirelessChannel",IDWirelessChannel,IDAttribute,
  [5,8],undef,
  [
   {"WCType",IDWCType,IDAttribType,
    [0,inf], IDString,[]}
  ]
 }
 {"IPAddress", IDIPAddress, IDIPAddr,
  [1,1], "192.168.0.1", []
 }
]}
```

Note that the ID of the entity remains the same; however, the new structure allows it to be further instantiated, hence the registered service can now spawn sub-services within the set cardinality limits.

VII. Conclusion

Object-oriented modeling artificially limits subtyping to the design time domain only. Although traditional industrial automation solutions used to live with fixed categorization of embedded device types, the new wave of SOA based automation frameworks openly embrace SoS concepts and thus open up the up till now closed world of available sensors and actuators. Hence, a corresponding modeling approach must be put in place in order to facilitate controlled dynamic subtyping that may be lead to higher flexibility without any incurred backdrop on type precision. In this paper, we have proposed a candidate solution which is based on a simple relaxation of a formula of the multi-level meta-modeling approach called DMLA. The applicability of the proposed solution has been showcased by a simple sensor example. The practical implementation of the solution is still work in progress, though it may later become a standard element of any model based implementation of the Service Registry used by the Arrowhead framework.

References

[1] F. Blomstedt, "Arrowhead Framework Cookbook," 2014.
[2] C. Atkinson és T. Kühne, „The Essence of Multilevel Metamodeling," The Unified Modeling Language. Modeling Languages, Concepts, and Tools, Volume 2185, pp. 19-33, 2001
[3] Z. Theisz, G. Mezei, „An Algebraic Instantiation Technique Illustrated by Multilevel Design Patterns," in MULTI@MoDELS, Ottawa, Canada, 2015.
[4] Z. Theisz és G. Mezei, „Multi-level Dynamic Instantiation for Resolving Node-edge Dichotomy," in Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development, Rome, Italy, 2016.
[5] Z. Theisz és G. Mezei, „Towards a novel meta-modeling approach for dynamic multi-level instantiation," in Automation and Applied Computer Science Workshop, Budapest, Hungary, 2015.
[6] E. Boerger and R. Stark, Abstract State Machines: A Method for High-Level System Design and Analysis, Springer-Verlag Berlin and Heidelberg GmbH & Co. KG, 2003.