Cooking up a MEAL:
Creating a Meta Enterprise Architecture Language

Graham McLeod
Promis Solutions AG, P O Box 384, Howard Place, 7450, South Africa
http://www.pro-mis.com graham.mcleod@pro-mis.com

Abstract. This paper presents the case for, and work on, the creation of a Meta Enterprise Architecture Language (MEAL), realised as a domain specific language for the definition, population, maintenance, manipulation, representation and analysis of enterprise architecture models and meta models. The English-like textual language is intended as a high level API to architecture modeling and repository services, both for internal use within tools and external use between tools. Requirements for such a language are identified and work on a proof of concept implementation using the Squeak dialect of Smalltalk is described. A subset of the language syntax is presented. It is intended that the language, after initial proof of concept, will be placed in the public domain and eventually become a standard.

Keywords: Enterprise Architecture, Enterprise Modeling, Meta Modeling, Architecture Language, Domain Specific Language, Application Specific Language, Model Exchange

1 Introduction

Enterprise Architecture (EA) seeks ways to model and represent information concerning business, processes, applications, services, information, technology and other domain elements. Frameworks and methods exist to assist with the development of architecture capabilities, models and artifacts within the enterprise. Examples include Zachman [1], TOGAF from the Open Group [2], IAF from Cap Gemini [3] and FEAF [4] from the US Government. Architecture notations (visual languages) are gaining ground as standard ways to represent architecture concepts and models. A recent successful example is Archimate [5], first developed at the Telematica Institute in the Netherlands, and now being integrated with the Open Group initiatives.

EA information and models are typically held in a repository and managed by tools which allow the capture, manipulation, querying, reporting and (sometimes) analysis of this information. Examples include ARIS from IDS Scheer [6], System Architect from IBM [7], EVA Netmodeler from Promis Solutions [8], etc. The way in which users interact with these tools is typically via graphical diagram editors or forms-based interfaces. Inter-operation between tools is achieved, usually with some
difficulty, by exporting and importing information in fairly low level mechanisms such as XML and XMI [9].

The author believes that there are potential advantages to development of a Domain Specific Language (DSL) [12] to facilitate the definition, capture, maintenance, querying, analysis and output of EA elements and models. Such a language would provide an English-like means to interact with tools and repositories. It could also serve as a high level API to allow higher levels of automation and integration of EA information into other processes. It can be used within a tool to provide a high level language for the rapid and reliable development of business logic and processes to accomplish EA related tasks. Finally, it can provide a higher level language for the exchange of models between tools. Since the implementation is anticipated to provide an interactive usage mode and interpretive execution of requests, it may be better to call it an Application Specific Language (ASL).

It is intended that the resulting language would be placed in the public domain and eventually become a standard. An interesting recent development is the founding of the Open Model Initiative [24] which promotes collaboration between enterprise and conceptual modelers and the sharing of models placed in the public domain. The proposed language could facilitate such activities.

The paper describes early work in conceiving the language, a suitable syntax and a prototype to explore and test the ideas developed in the open source Squeak implementation of Smalltalk [13].

The Need for a Domain Specific Language

An obvious question is: Why do we need another language? The drivers that have led the author to pursue this route are discussed below.

Existing mechanisms for the successful exchange of EA model information between tools are problematic and have limitations. The primary vehicle available for such exchange is the Meta Object Facility/XML Metadata Interchange (MOF/XMI) standard from the Object Management Group. The difficulties of achieving successful transfer are well documented by Alanen and Porres [11] who point out that the variants in versions for XMI/MOF and UML result in 28 “legal” encoding variants. Practical industry experience confirms that tools can be compliant without producing files that are intelligible by other compliant tools. Even if the transfer is achieved successfully, this standard only deals with the transfer or synchronization of data and models. It does not address any operations pertinent to the domain beyond the simple add, change or delete operations on basic data structures. The related Diagram Interchange standard [14] has seen little implementation by vendors.

Another potential candidate would be the Web Ontology Language (OWL), which provides a fairly rich facility to define classes, relationships, ontologies, constraints
and assertions over the created models. OWL has a more accessible syntax than the XMI approach, although it can also be serialised to XML. Unfortunately, OWL is not widely supported in modeling tools, so its usefulness as an interchange format is somewhat restricted.

Archimate is an emerging standard EA Language, but it is a visual language (or notation). It is a good way to represent enterprise architecture information across the business, application, information and technical domains and supports service orientation well. We would regard Archimate as one (albeit a good one) of several possible representations of the semantics held in the meta model and repository of an EA tool.

None of OWL, XMI or Archimate offer operations relevant to the architecture domain and the work of enterprise architects. An ASL can offer these, in addition to other benefits. According to van Deursen [12], DSLs (and hence ASLs) offer the following benefits:

• Allow solutions to be expressed in the idiom and at the level of abstraction of the problem domain. Consequently, domain experts themselves can understand, validate, modify, and often even develop DSL programs
• Programs are concise, self-documenting to a large extent, and can be reused for different purposes
• Enhance productivity, reliability, maintainability, and portability
• Embody domain knowledge, and thus enable the conservation and reuse of this knowledge
• Allow validation and optimization at the domain level
• Improve testability

Our own experience over a period of nine plus years implementing advanced enterprise modeling tools and repositories leads us to concur. In fact, we have elements of an EA ASL within the current toolset as an internal “business transaction” layer within a Model View Controller architecture [15]. More domain specific functionality is implemented in the Model layer. Our plans call for the refactoring of this functionality into a coherent Architecture Language Layer, providing both an internal and external API.

An increasing number of frameworks and representations used in the EA discipline (Zachman, TOGAF, IAF, FEAF, Archimate, ...) and increased exposure in business and to executives, requires that EA information be easily mapped to multiple forms of input, output and representation. The latter will include text (English and more formal languages, such as XMI, XML, RDF, SQL, CSV); Diagrams comprising vector and raster images; Matrices; Compound Documents and Presentations. An ASL can provide a good abstraction from the details of the repository while providing a *lingua franca* for the representation layers. It can also serve as a batch or interactive application programming interface (API) for the exchange of data with other tools, or the inter-operation of EA tools in a larger environment.
Requirements for an EA ASL

Based on the functionality in our current product, our experience and previous research [16] as well as literature, including the Enterprise Architecture Tool Selection Guide from IFEAD [17] and bearing in mind the definition of ASL properties from [12], we compiled a list of requirements for the MEAL. These include functional and non-functional requirements. We also compiled a number of Test Cases for validation. Clarifying the scope and naming, we defined the terms in the name as follows:

- **Meta** - Must be able to extend concepts/types and use resulting objects naturally in the language
- **Enterprise** - Should be scalable and capable of industrial scale use across at least the business, process, application, service, information and technology domains as well as the risk, cost, quality aspects
- **Architecture** - Address things, interfaces, relationships, properties, criteria/requirements, combinatorial rules and other concepts relevant to the architecture domain such as choreography, orchestration, sequence
- **Language** - In the first instance an application specific textual language. English-like and suitable for use by humans and automatic parsing and execution. Also as an API to an Architecture Repository. Capable of (meta) describing mapping to other languages (e.g. Visual, Formal semantics (e.g. OWL)). The language should be high level, English-like (not an XML family language) and declarative so that users would not be required to know implementation details or have algorithmic programming skills

Functional Requirements:

*Key concepts* that the language must be able to express are detailed below. These were informed by requirements, as well as prior work on an advanced repository meta model [18]:

- **Types** (equivalent to Classes) and instances of types
- **Properties** of types and instances
- **Relationships** between types and instances
  - Associations with cardinality constraints: 1 to 1; 1 to Many; Many to Many; optionality. Associations named in each direction
  - Own type relationships: Hierarchy; Dependence; Sequence etc.
  - Containment (holding an object as a property on another)
  - Inheritance: Single inheritance between types and sub-types
  - Roles: where a role is a specialization of a base type via containment of an instance of the base type. Roles allow elegant resolution of many of the problematic issues of multiple inheritance
  - Realisation or abstraction: Logical → Physical as one example
  - Category: where instances of one type are used to group instances of another
• **Association** properties: similar to the idea of an association class holding properties of an association in UML

• **Collections** of instances: with capabilities for management of arrays, sets, ordered and sorted collections

• **Representation**: defines how to render types, relationships and instances. Includes options for: Text, XML, Bitmap, Vector Symbol, UI Control etc.

• **Model Types**: define a kind of model via specifying a subset of the meta model and the representation of the relevant concepts. At two levels: Logical Model Type is purely the meta model subset; Physical Model Type maps to the representation and medium. There can be multiple physical model types per logical model type. A single physical model type could also apply to similarly structured logical model types that involve different meta model concepts.

• **Models**: A container for objects that are part of a model which conforms to one logical model type. Objects can be part of multiple models

• **Identity**: to uniquely name and subsequently refer to objects

• **Filter conditions**: These allow filtering of content allowed by security permissions to satisfy current user viewing or analysis requirements

• **Contexts**: Flexible, a container for objects relevant to a particular purpose — e.g. Business Unit; Project; Company

• **Domains**: equivalent to context for meta model. E.g. Business, Process, Application, Information, Project Management, Technology

• **Ownership**: a relationship between “owner” instances of a type such as “Role” and type definition in the meta model
  ◦ Permissions: what a designated user is allowed to view or do to content and to meta definitions

• **Requirements** and Goals that must be met

• **Criteria** by which architecture elements can be evaluated

• **Status**: e.g. Private, draft, published, retired..

• **Versions** (Current, future, past, named..)

**Key Operations** that the language must support include:

• **Creation and maintenance of types**, relationships, properties

• **Creation and maintenance of instances**, relationships, properties

• **Management of users** and the system

• **Importation** of data from common formats (at least CSV, XML)

• **Exportation** of data to common formats (as above)

• **Queries** on meta and instances

• **Reporting** on meta and instances

• **Analysis** on meta and instances, including:
  ◦ Comparison
  ◦ Summary
  ◦ Derivation of statistics
  ◦ Application of user defined algorithms
Protocol for definition of user defined techniques per object type e.g.

[Object] [analysisType:] [input parameter or list] returning: [output parameter or list]

- Document composition and incorporation of styles
- Generation of information in variety of formats including:
  - Individual Item
    - Text: Natural language; XML; CSV; XMI; SQL
    - Form with hyperlinks for relationships
  - Groups of Items
    - List
    - Tree
    - Matrix (Items and Relationships)
    - Table (Items and Properties)
    - Visual: e.g. Mindmap; Graph; Composite Document
- Provide service / consume service to allow for easy integration with other products
- Definition of combinatorial rules: These should allow determination of compatibility between two architecture elements. E.g. An application system is compatible with a platform if its object code can run there and its required services (e.g. Database, transaction monitor) are available there
- Transaction demarcation: to ensure system and repository integrity when related updates are performed
- Tracking changes and auditing to facilitate governance, security and recovery (more a function of the implementation than the language per se, but may need facilities at least to query such a log)

Non-Functional Requirements

To meet the goals of becoming an industrial strength and useful language, the MEAL and an implementation should meet certain non-functional requirements, including:

- Scalability: The system should be able to handle large meta models (100s to 1000s of types); large collections of instances (up to 1 000 000 instances per type); unlimited model types and models; and large models (1000s of instances per model). It should provide for implementation possibilities which can support large user communities (100s ) concurrently.
- Ease of learning and use: The language should be intuitive (as far as possible) for its target community, comprising architects and modelers.
- Extensibility: It should be possible to extend the language in well defined and manageable ways without destroying the utility of the core language or losing other important benefits. Examples of desirable extensions include:
  - Support for additional property data types. A protocol definition should allow new data types to be added to the system by defining an appropriate class
User defined operations. Users should be able to define operations to perform on architecture elements or types. There should be a clearly defined mechanism for the specification of these and naming conventions to ensure that they integrate with the language syntax in as seamless a way as possible.

- **Openness**: The language should provide for the invocation of external services (and hence obtaining input information) via well supported industry standard service protocols (e.g., SOAP[27]). The language should support external queries made using similar service protocols.

### Test Cases for Validation

We wanted a benchmark to decide when the MEAL could realistically claim to be a useful language for architecture work. The following were defined as test cases that should be met:

- Describe elements of business, process, application, information and technical architecture
- Describe aspects of risk, cost, quality, governance (these are cross cutting concerns which would be catered for by properties across a domain within the meta model)
- Describe examples from the inspired EA Frameworks [25], TOGAF and Archimate
- Perform all maintenance operations on meta model, content models and model representation mappings
- Add value with specific architecture oriented high level operations such as:
  - checkCompatibilityWith:
  - checkComplianceWith:
  - deriveScenarioFrom:
- Demonstrate extension for a basic property data type
- Demonstrate extension of operations by user

### Prototype Implementation Considerations

It was decided to prototype the language using an evolving test bed. This was to allow rapid and early validation of the thinking and ideas in a practical manner. We have long experience of the Smalltalk language (IBM Visual Age and Squeak) and considered this a potentially suitable environment for the creation of the MEAL ASL. We found support for this in the literature, including the successful implementation of a variety of DSLs in Smalltalk [19] [20] [21]. Another notable DSL based upon Smalltalk syntax, is the OMG’s Object Constraint Language (OCL) [22]. Our experience is that Smalltalk lends itself to a style of programming which uses DSL concepts: Adele Goldberg, one of the Smalltalk development team members, concurs [23]:
“When the language is used to describe an application system, the developer extends Smalltalk, creating a domain-specific language by adding a new vocabulary of language elements while maintaining the same semantics and syntax.”

Due to our desire to retain an “open” orientation for the language and related work, we selected the Squeak open source Smalltalk environment for prototyping purposes. This has an active community, good development tools and is available across a very wide set of platforms.

The full MEAL might be said to be ambitious and have many “courses”. The goal of the prototype is to prove the concepts and refine the thinking. We decided to simplify the initial “course”/phase as follows:

- We would work in a single user environment
- Persistence would be provided simply via the standard Smalltalk “image” i.e. saving the entire state of the system to disk
- Property data types would be limited to string, decimal and web-friendly bitmap images (jpeg, gif, png): Patterns and protocols would be created, however, to define how a data type is added to the language
- We would develop the language and underlying classes only – no graphical user interface
- Web services (SOAP) support would be ignored for now
- A very limited set of operations would be created, but patterns would be established for how these are defined
- Import and Export would be limited to CSV for now
- We would build “inside out” i.e. The core meta meta model classes first, followed by the meta model and then instance models. Other layers will be added in future, including security, filters, representations, etc.

Language Syntax

One of the strengths of Smalltalk is its very concise and consistent syntax. Everything in the language is an object (including classes, the compiler, elements of the user interface, IDE browsers...). Objects are queried or asked to perform operations (methods) via sending of messages. Messages are composed of keywords, which can be followed by parameters. It is thus fairly easy to compose English-like statements which are valid Smalltalk code. To achieve this, it is vital to choose class and method names carefully to express the relevant domain concepts.

Example of legal Smalltalk statements:

1. With built in class libraries:

   ```smalltalk
   Point x: 10 y: 20.  “sets the x and y coordinates of Point”
   ```
2. With domain extensions:

   Sale amount: 100 customer: MrsJones.

   "sets the sale amount to 100 and the customer to Mrs Jones"

The latter assumes that we have a Sale class and have defined the method amount:customer: there. Note that Smalltalk uses double quotes to delimit comments.

We have tried to follow the natural syntax as closely as possible, while creating a high level domain language which would be comfortable for the majority of architects and modelers. To keep the implementation effort small and reuse as much as we can (which also enhances reliability), we have tried to use as much of the Squeak provided mechanisms as we can. Specifically:

- Types become Smalltalk Classes, based on a parent class Type which provides behaviour and common properties for all types
- Manipulation of meta model is achieved by directing messages to Classes
- Manipulation of instance model items is achieved by directing messages to the items (instances of the meta model classes)
- Class objects manage collections of instances. This can be extended in future to provide persistence. One extension is the case of superclasses, where the allInstances mechanism is modified to return all instances of the receiver, as well as any subclasses. Thus requesting allItems from the type Person, would also return all instances of its subtype Employee.
- Property data types become abstract data types (also Smalltalk Classes, but in a different hierarchy). These need behaviour to serialise themselves, present themselves in various forms, provide validation rules for their instances etc.
- MEAL language operations become behaviour on classes. Method names are carefully chosen to create a natural syntax
- Standard Smalltalk IDE tools have been used to create the classes and methods and to test them (Class Browser, Workspace, SUnit)
- The dynamic ability of Smalltalk to compile and link new code (including new class definitions and methods) into the runtime image without disrupting use of the system is exploited to allow interactive runtime extensibility of the language

Syntax Examples from the resulting language are provided below. Comments in quotes describe what the code achieves:

"Declaring a new Type in the Meta Model"

   Person:=Type new.
"Adding typed properties to the Type - also adds the get and set methods and validation of data in these"

Person addProperty: 'name' whichIsA: 'String';
    addProperty: 'telephoneWork' whichIsAn: 'Integer'.

"Add a relationship between instances of two types. This is an untyped relationship"

Person addRelationship: 'fulfills|played by' to: Role.

"Add a typed relationship between two types."

Task addRelationship: 'requires|used by' whichIsA: 'dependency' to: Document.

"Add an own type relationship between instances of the same type. Note the relationship is also typed allowing special handling of hierarchies, containment, dependency etc."

Person addParentChildRelationship: 'manages|reports to'.

"Add a type which is a role of another. Role inherits properties and behaviours, but may have unique identity in addition to unique properties and behaviours. Role can be destroyed without destroying the base object."

Person addRole: 'Customer'.

"Adding a property to the type Customer which is itself another object. This effectively allows embedding of rich data types."

Customer addProperty: 'account' whichIsAn: 'Account'.

"Creating a new Customer object, and filling in the property values"

JoeBloggs:= Customer new.

JoeBloggs name: 'Joe Bloggs'; telephoneWork: 5315404.

"Create an Account object and associate it with Joe Bloggs"
Acct1234:=Account new.

JoeBloggs account: Acct1234.

"Create a relationship between two Person objects.
Related object created on the fly"

JoeBloggs manages: (MaryStewart:=Person new).

"Get the list of items of a specified type and its subtypes"

|List |
List:=Person allItems.

"Get the list of all items of a specified type within a given context"

List:=Person allItemsWithin: 'Retail'.

"Export the items of a type to a CSV formatted output"

| Table |
Table:=Person allItems asCSV.

"Perform architecture specific operations on items"

(SAPAccountsPayable isCompatibleWith: Oracle9i)
"returns true or false, whereupon we can decide what to do."

**Project Status and Preliminary Conclusions**

The project is still at an early stage, with the prototype under construction and evaluation. Nevertheless, the early results are encouraging. If one considers the sample code provided earlier, it amounts to some 17 lines of code, plus comments. In this brief amount of ASL, we have been able to achieve the following:

- Define a fragment of a meta model, including types, properties and relationships
- Instantiate new objects (items) conformant with the meta model
- Embed an item within another as a “rich” property
- Relate items
- Retrieve items of specified types, including filtering by context
- Export items to a useful tabular format
• Perform an architecture specific operation on relevant items

Further Work

Further work is intended to extend the MEAL to the other capabilities and scope discussed under requirements. This will proceed “bottom up”, building the meta modeling capability, the modeling capability, the representation capability, the import and export/generation capabilities, the persistence and logging capabilities and the user management and security capabilities in sequence, followed by the use of and provision of external services.

Challenges are still faced in the area of contexts and namespaces to ensure that items are uniquely named. It is relatively easy to achieve this within a given context and there are a variety of schemes available for doing it on a more global scale (e.g. XML style name spaces), but there is a challenge to make this formality intuitive for the language user, or at least transparent. The approach used in OCL may be an attractive option for adoption.

We are investigating the concepts of traits, derived from earlier work on mixins, as implemented in Squeak 3.9 onwards. Traits [26] allow the definition of blocks of functionality which can be reused easily by inclusion independent of the normal inheritance hierarchy. They provide many of the benefits of multiple inheritance, while avoiding the difficulties of clean implementation. The mechanism may assist us in keeping the language implementation very compact, while at the same time ensuring that the user facing syntax is as easy to use as possible. In particular, traits may assist with the implementation of translation for representation and the addition of user defined extensions of operations to meta model types.

We invite comments on the ideas and approaches presented in this paper and would like to collaborate with others who share an interest in this or a related area of investigation.

References

11. Alanen, Marcus; Porres, Ivan: Model Interchange Using OMG Standards,
12. van Deursen, Arie; Klint, Paul; Visser, Joost: Domain-Specific Languages,
    http://homepages.cwi.nl/~paulk/publications/Sigplan00.pdf
15. McLeod, Graham: Pamela: A Proto-pattern for Rapidly Delivered, Runtime Extensible
    FinalEditedFormatted.pdf
    comprehensive web-based solution,
17. Institute for Enterprise Architecture Developments; Enterprise Architecture Tool Selection
    %20Tools/Enterprise%20Architecture%20Tool%20Selection%20Guide%20v4.2.pdf
18. McLeod, Graham: (Meta) Meta Model Extensions for Manageability of Large Scale
    Collaborative Modeling, Proceedings EMMSAD June 2008, Mont Pellier, France. Available
    Computer Science, Springer, Berlin. Available at:
    http://www.springerlink.com/content/p94k6g306bxh5753/
20. Gobelis, Kris: Can Domain-Specific Languages Benefit from Linguistic Symbiosis?, Vrije
    22.pdf
21. Kermarrec, Y; Zein, O; le Pors, E; Grisvard, O: Towards the use of requirements as valuable
    inputs for complex system design, Information and Communication Technologies: From
    Theory to Applications, 2008. ICTTA 2008. 3rd International Conference,
    http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=4530345
    10-14.pdf
    http://portal.acm.org/citation.cfm?id=226239.226260
24. Open Model Initiative: http://openmodels.wiwinf.uni-dues.de/
25. Inspired: The Inspired Enterprise Architecture Frameworks,
    Traits/
27. W3C: Simple Object Access Protocol (SOAP) 1.1, http://www.w3.org/TR/2000/NOTE-
    SOAP-20000508/