Delay-Tolerant Networking E. Birrane
Internet-Draft E. DiPietro
Intended status: Informational D. Linko
Expires: January 1, 2019 Johns Hopkins Applied Physics Laboratory
June 30, 2018

AMA Application Data Model
draft-birrane-dtn-adm-03

Abstract

This document defines a physical data model that captures the information necessary to asynchronously manage applications. This model provides a set of common type definitions, data structures, and a template for publishing standardized representations of model elements.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on January 1, 2019.

Copyright Notice

Table of Contents

1. Introduction

The Asynchronous Management Architecture (AMA) [I-D.birrane-dtn-ama] defines a concept for the open-loop control of applications (and protocols) in situations where timely, highly-available connections cannot exist amongst managing and managed nodes in a network. While the AMA provides a logical data model, it does not include the detailed information necessary to produce interoperable data models.

1.1. Scope

This document defines a physical data model suitable for managing applications in accordance with the AMA. This physical model is termed the Asynchronous Management Model (AMM) and consists of the data types and data structures needed to manage applications in asynchronous networks.

This document also provides a template, called the Application Data Model Template (ADMT), for the standardized representation of application-specific instances of this model. Using the types and structures defined by the AMM, individual applications can capture their unique, static management information in documents compliant with the ADMT. These application-specific documents are called Application Data Models (ADMs).

The AMM presented in this document does not assume any specific type of application or underlying network encoding. In order to

communicate model elements between AMA Agents and Managers in a network, the model must be encoded for transmission. Any such encoding scheme is outside of the scope of this document. Generally, the encoding of the model is a separate concern from the specification of data within the model.

Because different networks may use different encodings for data, mandating an encoding format would require incompatible networks to encapsulate data in ways that could introduce inefficiency and obfuscation. It is envisioned that different networks would be able to encode ADMs in their native encodings such that the translation of ADM data from one encoding to another can be completed using mechanical action taken at network borders.

Since the specification does not mandate an encoding format, the AMM and ADMT must provide enough information to make encoding (and translating from one encoding to another) an unambiguous process. Therefore, where necessary, this document provides identification, enumeration and other schemes that ensure ADMs contain enough information to prevent ambiguities caused by different encoding schemes.

2. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

3. Terminology

Note: The terms "Actor", "Agent", "Externally Defined Data", "Variable", "Constant", Control", "Literal", "Macro", "Manager", "Operator", "Report", "Report Template", "Rule", "State-Based Rule", "Table", "Table Template", and "Time-Based Rule" are used without modification from the definitions provided in the [I-D.birrane-dtn-ama].

Additional terms defined in this document are as follows.

- Application - A software implementation running on an Agent and being managed by a Manager. This includes software that implements protocol processing on an Agent.

- Application Data Model (ADM) - The set of statically-defined data items necessary to manage an application asynchronously.

- Application Data Model Template (ADMT) - A standard format for expressing predefined data items for an application.

- AMM Resource Identifier (ARI) - A unique identifier for any AMM object, syntactically conformant to the Uniform Resource Identifier (URI) syntax documented in [RFC3986] and using the scheme name "ari".

- ADM Namespace - A moderated, hierarchical taxonomy of namespaces that describe a set of ADM scopes. Specifically, an individual ADM namespace is a specific sequence of ADM namespaces, from most general to most specific, that uniquely and unambiguously identify the namespace of a particular ADM.

- Operational Data Model (ODM) - The operational configuration of an Agent. This includes the union of all ADM information supported by the Agent as well as all operational, dynamic configuration applied to the Agent by Managers in the network.

4. Data Modeling Concept of Operations

In order to asynchronously manage an application in accordance with the [I-D.birrane-dtn-ama], an application-specific data model must be created containing any predefined management information for that application. This model is termed the Application Data Model (ADM) and forms the core set of information for that application in whichever network it is deployed. The ADM syntactically conforms to the ADMT and uses the data structures and types that comprise the AMM.

The information standardized in the ADM represents static configurations and definitions that apply to any deployment of the application, regardless of the network in which it is operating. Within any given network, Managers supplement the information provided by ADMs with dynamic definitions and values. The operational configuration of the network is the union of all supported ADMs and all Manager-defined dynamic configurations. This is termed the Operational Data Model (ODM).

The relationships amongst the AMM, ADMT, and ADM are illustrated in Figure 1.

Data Model Relationship

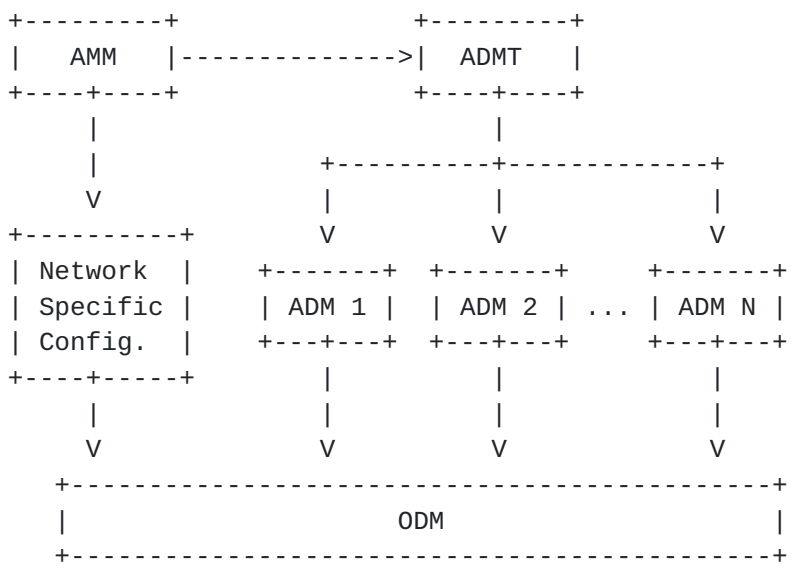


Figure 1

In this figure, AMM data types and structures form the common elements of the management model used by both ADMs and network specific configurations. Together, the set of static information provided by the union of all supported ADMs with the set of operator-specified dynamic AMM objects, forms the operational data model used to manage the network.

5. Asynchronous Management Model (AMM)

This section describes the Asynchronous Management Model, which is the set of objects used to implement the logical data model provided by the AMA. This section also provides additional information necessary to work with this model, such as data type specifications, identifier constructs, and naming conventions.

5.1. The AMM Resource Identifier (ARI)

Every object in the AMM must be uniquely identifiable, regardless of whether the item is defined formally in an ADM document or informally by operators in the context of a specific network deployment. The AMM Resource Identifier (ARI) uniquely identifies AMM objects.

There are three components to the ARI: namespaces, object names, and parameters. This section defines each of these components, discusses special cases, and presents a string canonicalization of these identifiers, with examples.

5.1.1. Namespaces

AMM objects exist within unique namespaces to prevent conflicting names within network deployments, particularly in cases where network operators are allowed to define their own object names. In this capacity, namespaces exists to eliminate the chance of a conflicting object name. They MUST NOT be used as a security mechanism. An Agent or Manager MUST NOT infer security information or access control based solely on namespace information.

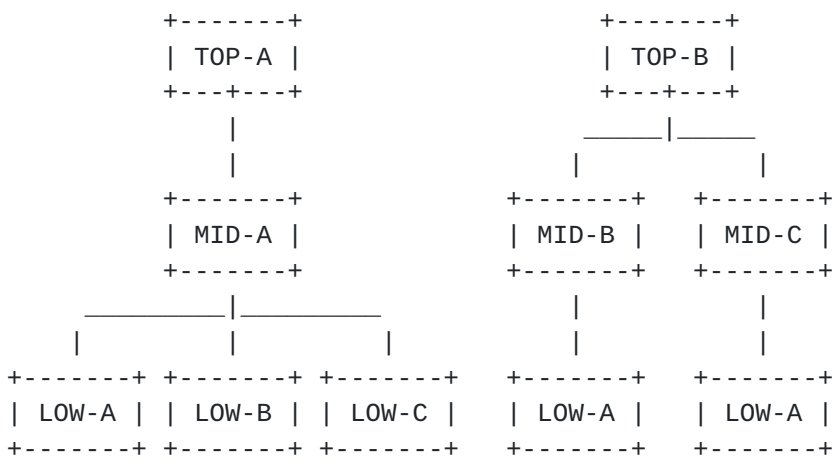
The AMM defines three ways to identify namespaces for AMM object names: Moderated Namespaces, Anonymous Namespaces, and Issuer Namespaces.

5.1.1.1. Moderated Namespaces

The most effective way to ensure the uniqueness of an AMM Object is to name it in the context of a moderated namespace. These namespaces are assigned by an overseeing organization as part of a maintained namespace registry.

Moderated namespaces are hierarchical, which allows the grouping of objects that share common attributes - for example, objects associated with related protocols may have protocol-specific namespaces that are grouped under a single encompassing namespace. Namespaces that are closer to a root node in the moderated hierarchy have broader scope than namespaces closer to leaf nodes in that hierarchy. There is no requirement that the namespace hierarchy be represented as a single tree structure; multiple root nodes are acceptable and likely to exist.

In a hierarchical model of namespaces, a particular namespace can be identified as the path to that namespace through the hierarchy. The expression of that path within an ADM is accomplished by listing each namespace along the path, separated by the tokenizing character "/". For example, consider the namespaces in the following figure.



Given this hierarchy, the following are all valid namespace representations.

TOP-A/

TOP-A/MID-A

TOP-A/MID-A/LOW-A

TOP-B/MID-B/LOW-A

TOP-B/MID-C/LOW-A

Moderated namespaces require resources to review and publish and are best suited for static AMM object definitions, such as those found in ADMs.

5.1.1.2. Anonymous Namespaces

It is possible for network operators to define AMM objects that are not associated with a namespace. In this case, a nil namespace can be defined. This special case is considered the use of an "anonymous" namespace.

Policy decisions as to whether anonymous namespaces are allowed in the system should be determined before network deployment. The use of an anonymous namespace greatly increases the chances of naming collisions.

5.1.1.3. Informal Namespaces

Network-specific configurations, as illustrated in Figure 1, are dynamic, ephemeral, not captured in published ADMs, and do not use moderated namespaces. Instead, AMM objects that comprise network-

specific configuration can be uniquely differentiated as a function of their "Issuer" and an issuer-specific "Tag".

An Issuer is any string that identifies the organization that is defining an AMM object. This value may come from a global registry of organizations, an issuing node address, a signed known value, or some other network-unique marking. Issuers MUST NOT conflict with known moderated namespaces, and Agents and Managers should not process Issuers that conflict with existing moderated namespaces.

A Tag is any string used to disambiguate AMM Objects for an Issuer. The contents of the tag are left to the discretion of the Issuer. Examples of potential tag values include an issuer-known version number or a (signed) hashing of the data item associated with the reference identifier.

5.1.2. Object Names

Object names are strings whose value is determined by the creator of the object. For those objects defined in accordance with the ADMT Template, the structure of the object name is given in Section 5.3.1.

5.1.3. Parameters

Parameterization is used in the AMM to enable expressive autonomous function and reduce the amount of traffic communicated between Managers and Agents. In the AMM, most objects can be parameterized and the meaning of parameterization for each object is described in detail in Section 5.3.

If there are two instances of an AMM object that have the same namespace and same object name but have different parameters, then those instances are both unique and the ARIs for those instances MUST also be unique. Therefore, parameters are considered part of an AMM object's identifier.

There are two types of parameters defined in the AMM: Formal and Actual parameters. The terms formal parameter and actual parameter follow common computer programming vernacular for discussing function declarations and function calls, respectively.

5.1.3.1. Formal Parameters

Formal parameters define the type, name, and order of the information that customizes an AMM Object. They represent the unchanging "definition" of the parameterized object.

Formal parameters MUST include type and name information and MAY include an optional default value. If specified, a default value will be used whenever a set of actual parameters fails to provide a value for this formal parameter.

5.1.3.2. Actual Parameters

Actual parameters represent the data values passed to a parameterized AMM Object. They "fulfill" the parameter requirements defined by the formal parameters for that object.

An actual parameter MUST specify a value and MAY specify a type. If a type is provided it MUST match the type provided by the formal parameter. An actual parameter MUST NOT include NAME information.

There are two ways in which the value of an actual parameter can be specified: parameter-by-value and parameter-by-name.

Parameter-By-Value
: This method involves directly supplying the value as part of the actual parameter. It is the default method for supplying values.

Parameter-By-Name
: This method involves specifying the name of some other parameter and using that other parameter's value for the value of this parameter. This method is useful when a parameterized AMM Object contains another parameterized AMM Object. The contained object's actual parameter can be given as the name of the containing object's parameter. In that way, a containing object's parameters can be "pass down" to all of the objects it contains.

5.1.3.3. Optional Parameters

In cases where a formal parameter contains a default value, the associated actual parameter may be omitted. Default values in formal parameters (and, thus, optional actual parameters) are encouraged as they reduce the size of data items communicated amongst Managers and Agents in a network.

5.1.4. Special Case: Literal Values

As defined in the AMA, Literal values are those whose value and identifier are equivalent. For example, the literal "4" serves as both an identifier and a value. When literal values are used in objects in the AMM, they are able to have a simplified identification scheme.

Because the value of a Literal object serves as its identifier, there is no need for namespaces, object names, or parameters. A literal can be completely identified by its data type and data value. Since Literals in the AMA are used to identify primitive data types, the type of a Literal identifier MUST be as described in Table 2.

5.1.5. String Canonical Forms

While there may exist multiple encodings of an ARI, to include the JSON encodings presented in Section 6 and other binary encodings in other specifications, this section defines a universal string representation of the ARI, as such a representation is helpful to express examples in this and other documents.

This representation is not prescriptive; other string encodings may exist that differ from the one used in this document.

5.1.5.1. General ARI String Representation

The String Canonical Form of the ARI is expressed as a Uniform Resource Identifier (URI), as documented in [RFC3986]. A URI is syntactically decomposed into a scheme name and a scheme-specific part. The set of known scheme names is moderated by IANA. The scheme-specific part of the URI is dependent on the scheme name.

The scheme name of the ARI is "ari". The scheme-specific part of the "ari" scheme follows the format:

ari:/<Namespace>/<ObjectName><(Parameters)>

With the string representation of each scheme given as follows.

Namespaces

Namespaces are represented as "/" separated lists, with individual namespace types represented as follows:

* Moderated namespaces are listed in order from the most general namespace to the most specific namespace. For example: "GENERAL/MIDDLE/SPECIFIC/".

* Anonymous namespaces are empty and are represented as "/".

* Informal namespaces follow the general pattern of moderated namespaces - starting with the general Issuer followed by the more specific issuer tag. For example: "Issuer/Tag". In cases where the Tag is omitted, then the representation is simply "Issuer/".

Object Names
: The object name is a string as specified in Section 5.3.1.

Parameters
: If present, parameters are represented as a comma-separated string enclosed in parenthesis. Different types of parameters are represented as follows.

  * Formal parameters follow the pattern <type> <name> and if there is a default value, it is represented by the substring "= <value>".

  * Actual Parameters-By-Value are represented as the string encoding of their value.

  * Actual Parameters-By-Name are represented as the name of the parameter enclosed in angle brackets.

  Note: If an actual parameter is missing for a formal parameter that has a default value, then the ARI string MUST have a blank space where the actual parameter would have been. This missing parameter will also have a comma, separating it from other actual parameters in the ARI string.

5.1.5.1.1. Shortform Encoding

In cases where a default namespace can be assumed (for example, in the context of an ADM with a defined namespace) the prefix ari:/Namespace/ can be omitted.

5.1.5.2. Literal String Encoding

The string representation of a Literal ARI is much simpler and consists of simply the data type of the Literal followed by the value, as follows:

"(type) value"

5.1.6. Examples

The ARIs for the following sample AMM objects are encoded in Table 1. Note that these examples are for the identifiers of AMM objects, not their entire definition.

- The number of bytes received by an Agent, defined in the N1/N2 namespace and called num_bytes.

- The number of bytes received through an interface, called num_bytes_if, which takes a single string parameter named "if_name" with a default value oth "eth0".

- An anonymous, operator-defined object named "obj1" which takes two unsigned integer parameters, n1 and n2, with default values of 3 and 4, respectively.

- The typed, Literal value of 4.

ARI String	Description
"ari:N1/N2/num_bytes"	Unparameterized num_bytes object in the N1/N2 namespace.
"num_bytes"	Shortform encoding where the N1/N2 namespace can be assumed.
"num_bytes_if(String if_name)"	Formal parameter definition of num_bytes object that accepts a string interface name.
"num_bytes_if(String if_name=eth0)"	Formal parameter definition of num_bytes object that accepts a string interface name with a default value.
"num_bytes_if()"	Actual parameter using the default value of eth0.
"num_bytes_if(eth0)"	Actual parameter of eth0.
"ari:/obj1(Int n1 = 0, Int n2 = 3)"	Formal parameter of object obj1 in anonymous namespace taking 2 default parameters.
"ari:/obj1(,)"	Actual parameter using the default values of 0 for n1 and 3 for n2.
"ari:/obj1(, 4)"	Actual parameter using the default value of 0 for n1.
"ari:/obj1(4,)"	Actual parameter using the default value of 3 for n2.
"ari:/obj1(4,4)"	Actual parameters provided for all obj1 parameters.
"ari:/obj1(<input>,4)"	Actual parameters provided for all obj1 parameters, with the value of the first parameter taken from some other parameter named "input".
"(UINT) 4"	The Literal value 4 interpreted as a 32-bit unsigned integer.

Table 1

5.2. AMM Type Definitions

This section describes the type definitions used by the AMM.

5.2.1. Primitive Types

The AMM supports a series of primitive types as outlined in Table 2.

Type	Description
BYTE	unsigned byte value
INT	32-bit signed integer in 2's complement
UINT	32-bit unsigned integer in 2's complement
VAST	64-bit signed integer in 2's complement
UVAST	64-bit unsigned integer in 2's complement
REAL32	Single-precision, 32-bit floating point value in IEEE-754 format.
REAL64	Double-precision, 64-bit floating point value in IEEE-754 format.
STRING	NULL-terminated series of characters in UTF-8 format.
BOOL	A Boolean value of FALSE (whose integer interpretation is 0) or TRUE (whose integer interpretation is not 0).

Table 2: Primitive Types

5.2.2. Derived Types

A derived typed is a primitive type that is interpreted with special semantics. The AMM supports the following derived types.

5.2.2.1. Byte String

A Byte String is a specialization of the String primitive data type used to store binary data using base64 encoding as defined in [RFC4648].

5.2.2.2. Time Values (TV) and Timestamps (TS)

A Time Value (TV) is a specialization of the String primitive data type whose time interpretation is as given in this section. There are two "types" of time representations within the AMM: relative times and absolute times.

An absolute time represents an instant in time. It MUST be formatted as a date-time in accordance with [RFC3339].

A relative time is defined as the amount of time after an instant in time. A relative time MUST be formatted as a full-time in accordance with [RFC3339]. Relative times have advantages over absolute times: they do not require time to be synchronized across Agents and Managers, and they are more compact in their representation. For example, expressing the semantics "run control_one 10 seconds after receiving it" or "run control_two 20 seconds after running control_one" is more appropriate using relative times than absolute times. The initiating event of a relative time MUST be unambiguously defined in the context using the time value.

As a practical matter, encodings of relative times MAY impose a limit of no more than 17 years of relative time, which corresponds to roughly 29 bits of information and is considered well past an upper bound of efficiency for using a relative time versus an absolute time.

An absolute time may be differentiated from a relative time based on whether the time specification is a date-time or a full-time.

For example, "00:00:10Z" is a relative time representing 10 seconds after an initiating event. "2019-01-01T08:00:00Z" is an absolute time that refers to 8am, Tuesday January 1st, 2019.

A Timestamp (TS) represents a specific point in time when an event occurred. As such, it MUST be represented as an absolute time.

5.2.2.3. Type-Name-Value (TNV)

A Type-Name-Value (TNV) is a three-tuple of information that describes a typed, named value in the AMM. Since the length of a data value is a matter of encoding, there is not an explicit length field present for the data value; it is assumed that any encoding scheme either explicitly encodes length or that the length is self-delineating in the encoding.

- Type - The strong typing for this value. Types MUST be one of those defined in Section 5.4.

- Name - A unique identifier for this value.

- Value - The value of the data item.

5.2.2.4. User-Specified Derived Types

Individual ADMs and network operators may derive other types that specialize the types provided by the AMM. When doing so, AMM data types MUST be used to capture the specialization and any user-specific verification or validation MUST occur in user-specific implementations on Agents and Managers.

5.2.3. Collections

AMM objects, or parameters associated with those objects, often need to represent groups of related information. Since the AMM is strongly typed, these groups of related information are represented by special data types called collections. AMM collections are ordered and may contain duplicate entries.

The AMM defines three typed collections that capture TNVs, ARIs, and mathematical expressions.

5.2.3.1. Type-Name-Value Collection (TNVC)

A Type-Name-Value Collection (TNVC) is an ordered array where each element of the array is a TNV.

TNVCs are often used to capture formal and actual parameters for AMM objects.

5.2.3.2. ARI Collection (AC)

An ARI Collection (AC) is an ordered set of ARIs.

ACs are often used when there exists a need to refer to multiple AMM objects as a single unit. For example, when defining a Report Template, the definition may have an AC that defines the ordered ARIs whose values constitute that report.

5.2.3.3. Expression (EXPR)

An Expression (EXPR) is a specialization of an AC where each ARI in the collection is either an operand or an operator. These operands and operators form a mathematical expression that is used to compute a numerical value.

Within an Expression, an operand MUST be an ARI with one of the following types: Literal, Constant, Externally Defined Data, or Variable. An operator MUST be an ARI of type Operator.

Since the Expression is an AC, there are no annotative constructs such as parenthesis to enforce certain orders of operation. To preserve an unambiguous calculation of values, the ARIs that form an Expression MUST be represented in postfix order. Postfix notation requires no additional symbols to enforce precedence, always results in a more efficient encoding, and post-fix engines can be implemented efficiently in embedded systems.

For example, the infix expression A * (B * C) is represented as the postfix A B C * *.

Expressions are often used when assigning values to a Variable or when calculating the state of the Agent in the context of a State-Based Rule.

5.3. Object Definitions

This section identifies the AMM Objects that instantiate the AMA logical data model and the processing required to support these objects at Agents and Managers in the network.

5.3.1. Common Object Metadata

Every object in the AMM includes a set of metadata providing annotative or otherwise user-friendly descriptive information for the object. This information may be used as documentation (for example, only present in ADMs and on operator consoles) and/or encoded and transmitted over the wire as part of a management protocol.

Metadata is not required to be unique amongst objects and individual encodings MAY choose to not encode metadata in cases where the information is not needed to uniquely identify objects. The metadata supported by the AMM for objects is as follows:

(STR) Name
: An object name is a string associated with the object, but does not constitute the sole identifier for the object. Names provide human-readable and/or user-friendly ways to refer to objects within a given context.

(STR) Description
: An object description is a string describing the purpose or usage of the object in a human-readable format. The description serves as documentation for the object and SHOULD

be the same regardless of how the object might be parameterized. For example, the description of a CTRL object should document the purpose of the CTRL in a way that is independent of the value of any particular parameter value passed to that CTRL.

5.3.2. Externally Defined Data (EDD)

Externally defined data (EDD), as defined in the AMA, represent data values that are computed external to the network management system. The definition of these values are solely defined in the context of an ADM; since their calculation exists outside of the network management system, they are not added or removed as part of the dynamic configuration of the network management system.

An EDD consists of an ARI, type, and a description, with the following caveats:

(ARI) Identifier
: This Identifier MUST be of type EDD and MAY be parameterized, particularly in cases where the specific computed value can be identified by an associative look-up, as discussed in Section 7.

(UINT) Type
: The data type of the EDD value MUST be specified as part of the EDD definition and this type MUST be one of the primitive data types defined in Table 2.

(STR) Description
: This represents the human-readable description of the EDD.

5.3.3. Constant (CONST)

Constants represent named basic values. Examples include common mathematical values such as PI or well-known Epochs such as the UNIX Epoch. Constants are defined solely within the context of ADMs. Constants MUST NOT be defined as part of dynamic network configuration.

Allowing network operators to define constants dynamically means that a Constant could be defined, removed, and then re-defined at a later time with a different value, which defeats the purpose of having Constants. Variables MUST be used instead of Constants for the purpose of adding new values to the dynamic network configuration.

A CONST is defined by its ARI, value, and description, with the following caveats.

(ARI) Identifier
: This Identifier MUST be of type CONST and MUST NOT be parameterized. Parameterizing a Constant implies that its value is dependent upon the set of parameters sent to it, which defeats the purpose of defining a constant value.

(TNV) Typed Value
: The value of a constant is the immutable value that should be used in lieu of the Constant ARI.

  This value is expressed as a TNV with the following requirements.

  * Type MUST be one of the primitive data types defined in Table 2.

  * Name MUSt be omitted as the CONST ARI defines the name for this value.

  * Value must be present and consistent with the data type for this CONST.

(STR) Description
: This represents the human-readable description of the CONST, as a string.

5.3.4. Control (CTRL)

A Control represents a predefined function that can be run on an Agent. Controls are not able to be defined as part of dynamic network configuration since their execution is typically part of the firmware or other implementation of the Agent outside of the context of the network management system.

Network operators that wish to dynamically execute functions on an Agent may use Macros, State-Based Rule, and Time-Based Rule instead.

Controls are identified by their ARI and their description, with the following caveats.

(ARI) Identifier
: This Identifier MUST be of type CTRL and MAY be parameterized in cases where the function executed by that Control takes parameters.
  When defined in the context of an ADM, the Control ARI MUST match the definition of a Formal Parameter list. This is because the ADM defines the Controls that can be invoked, but does not define any particular invocation of a Control.

When used as part of network operations, a Control ARI MUST match the definition of an Actual Parameter list. This is because when used operationally, a parameterized Control required parameters to be run.
In cases where a Control takes no parameters, the definition in the ADM document MUST be considered the definition of the Control and the presence of the same ARI in the context of an operational system MUST be seen as an invocation of that Control.

(STR) Description
This represents the human-readable description of the Control, as a string.

5.3.5. Macro (MAC)

Macros are ordered collections of Controls or other Macros. When run by an Agent, each ARI in the AC is run in order. A Macro may be defined as part of an ADM or as part of dynamic network configuration.

In cases where a Macro contains another Macro, implementations MUST implement some mechanism for preventing infinite recursions, such as defining maximum nesting levels, performing Macro inspection, and/or enforcing maximum execution times.

A Macro is defined by an ARI, a content definition, and a description, as follows.

(ARI) Identifier
This Identifier MUST be of type MAC and MAY be parameterized and, if so, the parameter may be passed-by-name to any parameterized elements within the Macro.

(AC) Definition
The Macro definition is modeled as an AC, where each ARI within the AC MUST be either a Control or a Macro.

(STR) Description
This represents the human-readable description of the Macro, as a string.

5.3.6. Operator (OP)

Operators represent user-defined mathematical functions implemented in the firmware of an Agent for the purpose of aiding the evaluation of Expressions.

The AMM separates the concepts of Operators and Controls to prevent side-effects in Expression evaluation (e.g. to avoid constructs such as A = B + GenerateReport()). For this reason, Operators are given their own structure type and Controls do not support a return value.

Because Operators represent custom firmware implemented on the Agent, they are not defined dynamically as part of network operations. Therefore, they may only be defined in an ADM.

An Operator is defined by its ARI, its resultant type, the number of operands, the type of operands, and a description, as follows.

(ARI) Identifier
: This Identifier MUST be of type OP and MUST NOT be parameterized. Much like Constants, Operators represent immutable mathematical functions. The operands of an Operator are not considered as "parameters" to the Operator.

(UINT) Out Type
: This is the return value of the Operator and MAY be different than the operand types accepted by the Operator. This type MUST be one of the primitive data types defined in Table 2.

(UINT) Num Operands
: This is the number of operands evaluated by the operator. For example, the unary NOT Operator ("!") would operate on a single operand. The binary PLUS Operator ("+") would operate on two operands. A custom operator to calculate the average of the last 10 samples of data would operate on 10 operands.

(TNVC) In Types
: This is the type information for each operand operated on by the Operator, modeled as a TNV Collection (TNVC). There MUST be one TNV in the TNVC for each operand, and each TNV MUST adhere to the following requirements:

  * The Type field MUST be present and MUST be one of the primitive data types defined in Table 2.

  * The Name field MAY be present to capture a semantic name for the operand.

  * The Value field MUST NOT be present.

(STR) Description
: This represents the human-readable description of the Operator, as a string.

5.3.7. Reports

A Report is a set of non-tabular, potentially nested data items that may be predefined in the context of an ADM, or defined dynamically in the context of a deployed network.

Reports are represented in two ways in the AMM: Report Templates and Reports. A Report Template defines the type of information to be included in a report, and a Report contains that information.

5.3.7.1. Report Template (RPTT)

A Report Template (RPTT) is the ordered set of data descriptions that describe how values will be represented in a corresponding Report. RPTTs can be viewed as a schema that describes how to interpret a Report; they contain no values and are either defined in an ADM or configured between Managers and Agents during network operations.

Since a RPTT may contain other RPTTs, implementations MUST implement some mechanism to prevent the definition of circular references.

RPTTs are defined by an ARI, the report template definition, and a description, as follows.

(ARI) Identifier
: This Identifier MUST be of type RPTT and MAY be parameterized and, if so, the parameter may be passed-by-name to any parameterized elements within the RPTT.

(AC) Definition
: The Report Definition is modeled as an AC, where each ARI within the AC MUST identify either a CONST, LIT, EDD, VAR, or other RPTT.

(STR) Description
: This represents the human-readable description of the Report template, as a string.

5.3.7.2. Report (RPT)

A Report (RPT) is a set of data values populated in conformance to a given data definition. Reports do not have an individual identifier - rather they are uniquely identified by their definition and the timestamp at which their data values were collected.

RPTs are defined by their associated template, the time at which the report was generated, and the individual entries in the report, as follows.

(ARI) Template Id
: This is the ARI of the object that defines the format of the report data values. This ARI MUST define an AMM object of type RPTT, EDD, or VAR, or CTRL.
If this ARI is parameterized, this ARI MUST include the actual parameters used in the generation of the report.

(TV) Generation Time
: This is the absolute time at which the report was generated by the Agent.

(TNVC) Report Entries
: This is the collection of data values that comprise the report. If the template for this report is an EDD, VAR, or CTRL then there MUST be one entry for this report. If the template is a RPTT, then there MUST be one entry for every item defined in the template.
Entries are modeled as a TNVC, with each TNV representing a report entry with the following requiremnts.

  * Type MAY be ommitted in cases where checking type safety is not required.

  * Name MAY be omitted in cases where a semantic name for the entry can be derived from the template.

  * Value MUST be present and consistent with the type for this entry from the associated template item.

5.3.8. State-Based Rule (SBR)

A State-Based Rule (SBR) specifies that starting at a particular time an action should be run by the Agent if some condition evaluates to true, until the action has been run a maximum number of times. When the SBR is no longer valid it MAY be discarded by the Agent.

Examples of SBRs include:

> Starting 2 hours from receipt, whenever Variable V1 > 10, produce a Report Entry for Report Template R1 no more than 20 times.
>
> Starting at some future absolute time, whenever Variable V2 != Variable V4, run Macro M1 no more than 36 times.

SBRs are defined by their ARI, start time, condition, maximum run count, action, and description, as follows.

(ARI) Identifier

This Identifier MUST be of type SBR and MUST NOT be parameterized.

(TV) START
: The time at which the SBR condition should start to be evaluated. This will mark the first evaluation of the condition associated with the SBR.

(EXPR) CONDITION
: The Expression which, if true, results in the SBR running the associated action. An Expression is considered true if it evaluates to a non-zero number.

(UVAST) COUNT
: The number of times the SBR action can be run. The special value of 0 indicates there is no limit on how many times the action can be run.

(AC) ACTION
: The collection of Controls and/or Macros to run as part of the action. This is captured as an AC data type with the constraint that every ARI within the AC represent a Control or Macro.

(STR) Description
: This represents the human-readable description of the SBR, as a string.

5.3.9. Tables

A Table is a named, typed, collection of tabular data. Columns within a table are named and typed. Rows within a table capture individual data sets with one value in each row corresponding to one column in the table. Tables are represented in two ways in the AMM: Table Templates and Table Instances.

5.3.9.1. Table Template (TBLT)

A table template identifies the strongly-typed column template that will be followed by any instance of this table available in the network. Table templates appear statically in ADMs and may not be created dynamically within the network by Managers. Changing a table template within an asynchronously managed network would result in confusion if differing template definitions for the same table identifier were to be active in the network at one time.

TBLTs are defined by an ARI, a set of column descriptions, and table metadata, as follows.

(ARI) Identifier
: This Identifier MUST be of type TBLT and MUST be of type TBLT, and MUST NOT contain parameters.

(TNVC) Columns
: A TBLT is defined by its ordered set of columns descriptions captured as a TNVC with each TNV in the collection describing a table column with the following requirements.

  * Type MUST be present and MUST be one of the primitive types defined in Table 2.

  * Name MAY be omitted in cases where a semantic name for the column can be derived from the template.

  * Value MUST NOT be present as a column does not contain data values.

(STR) Description
: This represents the human-readable description of the TBLT, as a string.

5.3.9.2. Table (TBL)

Tables are collections of data that MUST be constructed in accordance with an associated Table Template. Tables MUST NOT appear in the ADM for an application; they are only instantiated dynamically as part of the operation of a network.

TBLs are defined by their Table Template, the number of rows in the table, and the associated set of data values for each row.

(ARI) Template Id
: This is the ARI of the Table Template holding the column definitions for this table. This ARI MUST be of type TBLT and match a known Table Template.

(UINT) Number of Rows
: This is the number of rows in the table. A Table MAY have zero rows.

(TNVC) Rows Information
: Each row in a TBL is represented by a TNVC, with each TNV in the collection representing the value for a specific column with the following requirements.

  * Type MAY be present, when necessary to verify that elements in the row match the types of table columns.

* Name MUST NOT be present.

* Value MUST be present and compatible with the type of the associated column.

The number of TNVs in the collection MUST be equal to the number of columns defined for the Table Template.

5.3.10. Time-Based Rule (TBR)

A Time-Based Rule (TBR) specifies that starting at a particular start time, and for every period seconds thereafter, an action should be run by the Agent until the action has been run for count times. When the TBR is no longer valid it MAY be discarded by the Agent.

Examples of TBRs include:

> Starting 2 hours from receipt, produce a Report for Report Template R1 every 10 hours ending after 20 times.
>
> Starting at the given absolute time, run Macro M1 every 24 hours ending after 365 times.

TBRs are defined by their ARI, start time, period, maximum run count, action, and description, as follows.

(ARI) Identifier
: This Identifier MUST be of type TBR and MUST NOT be parameterized.

(TV) Start
: The time at which the TBR should start to be evaluated. This will mark the first running of the action associated with the TBR.

(TV) Period
: The time to wait between running the action associated with the TBR. This value MUST be a relative time value.

(UVAST) Count
: The number of times the TBR action may be run. The special value of 0 indicates the TBR should continue running the action indefinitely.

(AC) Action
: The collection of Controls and/or Macros to run by the TBR. This is captured as a AC with the constraint that every ARI within the AC represent a Control or Macro.

(STR) Description
: This represents the human-readable description of the TBR, as a string.

5.3.11. Variable (VAR)

Variables (VAR) may be statically defined in an ADM or dynamically by Managers in a network deployment. VARs differ from EDDs in that they are completely described by other known data in the system (either other VARs or other EDDs). For example, letting E# be a EDD item and V# be a VAR item, the following are examples of VAR definitions.

V1 = E1 * E2

V2 = V1 + E3

VARs are defined by an ARI, a type, an initializing expression, and a description, as follows.

(ARI) Identifier
: The type of this ARI MUST be type VAR, and the ARI MUST NOT contain parameters.

(UINT) Type
: This is the type of the VAR, and acts as a static cast for the result of the initializing EXPR. This type MUST be one of the data types defined in Table 2.
Note, it is possible to specify a type different than the resultant type of the initializing EXPR. For example, if an EXPR adds two single-precision floating point numbers, the VAR MAY have an integer type associated with it.

(EXPR) Initializer
: The initial value of the VAR is given by an initializing EXPR. In the case where the type of the VAR itself is EXPR, the initializer is used as the value of the VAR. In the case where the type of the VAR is anything other than EXPR, then the initializer EXPR will be evaluated and the result of that evaluation will be the initial value of the VAR.

(STR) Description
: This represents the human-readable description of the VAR, as a string.

5.3.12. Common Object Processing

This section describes the handling and exchange of AMM objects between Agents and Managers in a network.

Managers must:

- Store the ARI and definitions for both network-specific and ADM-defined AMM Objects.
- Send requests to Agents to add, list, describe, and remove custom AMM object definitions.
- Verify and interpret reports against report templates and tables against table templates when receiving these objects from an Agent.
- Encode ARIs in Objects to Agents, and decode ARIs from Agents.
- Provide actual parameters when sending parameterized objects to an Agent.

Agents must:

- Store the ARI for all ADM-defined AMM objects.
- Calculate the value of an AMM object when required, such as when generating a Report or evaluating an Expression.
- Implement Controls in firmware and run Controls and Macros with appropriate parameters when necessary in the context of Manager direction and Rule execution.
- Communicate "return" values from Controls back to Managers as Reports where appropriate.
- Persist custom AMM object definitions.
- Add, remove, list, and describe custom AMM objects as requested by Managers.
- Calculate the value of applying an Operator to a given set of operands, such as when evaluating an Expression.
- Populate Reports and Tables for transmission to Managers when required.

- Run the actions associated with SBRs and TBRs in accordance with their definitions.

- Calculate the value of VARs when required, such as during Rule evaluation, calculating other VAR values, and generating Reports.

5.4. Data Type Mnemonics and Enumerations

While the AMM does not specify any encoding of data model elements, a common set of enumerations help to ensure that various encoding standards can interoperate.

This section defines string (mnemonic) and integer (enumeration) mechanisms for referring to AMM data and object types. Data types are separated into 4 major categories:

Category	Range
AMM Objects Types	0x00 - 0x0F
Primitive Types	0x10 - 0x1F
Compound Types	0x20 - 0x2F
Reserved	0x30 - 0xFF

Type Categories and Ranges

Within each category, the type of information, it's mnemonic, unique enumeration value, and whether it is considered a numeric value for expression evaluation are listed.

5.4.1. AMM Objects

AMM Objects include the set of objects identifiable using the ARI construct. The type field of the ARI MUST be one of these values. AMM Objects MUST be identified as follows.

Structure	Mnemonic	Enumeration	Numeric
Constant	CONST	0	No
Control	CTRL	1	No
Externally Defined Data	EDD	2	No
Literal	LIT	3	No
Macro	MAC	4	No
Operator	OPER	5	No
Report	RPT	6	No
Report Template	RPTT	7	No
State-Based Rule	SBR	8	No
Table	TBL	9	No
Table Template	TBLT	10	No
Time-Based Rule	TBR	11	No
Variable	VAR	12	No
Reserved		13-15	No

5.4.2. Primitive Data Types

Primitive data include the basic set of objects that must be encoded to transfer AMM objects. All AMM objects are built from combinations of these primitive types. Primitive types MUST be identified as follows.

Basic Data Type	Mnemonic	Enumeration	Numeric
Boolean	BOOL	16	No
BYTE	BYTE	17	No
Character String	STR	18	No
Signed 32-bit Integer	INT	19	Yes
Unsigned 32-bit Integer	UINT	20	Yes
Signed 64-bit Integer	VAST	21	Yes
Unsigned 64-bit Integer	UVAST	22	Yes
Single-Precision Floating Point	REAL32	23	Yes
Double-Precision Floating Point	REAL64	24	Yes
Reserved		25-31	No

5.4.3. Compound Data Types

Compound data include combinations of primitive data types, to include collections. Compound types MUST be identified as follows.

Compound/Special Data Type	Mnemonic	Enumeration	Numeric
Time Value	TV	32	No
Timestamp	TS	33	No
Type-Name-Value	TNV	34	No
Type-Name-Value Collection	TNVC	35	No
AMM Resource Identifier	ARI	36	No
ARI Collection	AC	37	No
Expression	EXPR	38	No
Byte String	BYTESTR	39	No
Reserved - Protocol		40-47	No

5.4.4. Numeric Promotions

When attempting to evaluate operators of different types, an Agent may need to promote operands until they are of the correct type. For example, if an Operator is given both an INT and a REAL32, the INT should be promoted to a REAL32 before the Operator is applied.

Listing legal promotion rules is mandatory for ensuring that behavior is similar across multiple implementations of Agents and Managers. The listing of legal promotions in the AMM are listed in Figure 2. In this Figure, operands are listed across the top row and down the first column. The resultant type of the promotion is listed in the table at their intersection.

	INT	UINT	VAST	UVAST	REAL32	REAL64
INT	INT	INT	VAST	UNK	REAL32	REAL64
UINT	INT	UINT	VAST	UVAST	REAL32	REAL64
VAST	VAST	VAST	VAST	VAST	REAL32	REAL64
UVAST	UNK	UVAST	VAST	UVAST	REAL32	REAL64
REAL32	REAL32	REAL32	REAL32	REAL32	REAL32	REAL64
REAL64	REAL64	REAL64	REAL64	REAL64	REAL64	REAL64

Figure 2: Numeric Promotions

The AMM does not permit promotions between non-numeric types, and numeric promotions not listed in this section are not allowed. Any attempt to perform an illegal promotion SHOULD result in an error.

5.4.5. Numeric Conversions

Variables, Expressions, and Predicates are typed values. When attempting to assign a value of a different type, a numeric conversion must be performed. Any numeric type may be converted to any other numeric type in accordance with the C rules for arithmetic type conversions.

6. JSON ADM Template

This section provides an ADM template in the form of a JSON document and describes the JSON representation of AMM objects that MUST be used to populate this JSON ADM template.

It is not required that these JSON encodings be used to encode the transmission of AMM information over the wire in the context of a network deployment. It is also not required that only these JSON encodings be used to document ADMs and other AMM information. Since

the AMM is designed to allow for multiple encodings, the expression of ADMs in the provided JSON format is intended to support translation to other encodings without loss of information.

6.1. ADM Inclusion

ADMs expressed in conformance with this template are captured as individual JSON files. AMM Objects defined in one ADM template MAY refer to objects defined in another ADM template file. To enable type checking of these cross-ADM references, the ADM template supports the "uses" keyword to identify other ADM files that contain objects referenced in the current ADM file.

The syntax of the uses statement is as follows.

"uses":["file1","file2",...,"fileN"]

Where file_# represents a JSON-formatted ADM file defining a namespace used in this ADM file.

6.2. ADMT Object Collections

The JSON ADM Template is defined as a JSON object containing a series of arrays - one for each type of information specified in the template. There are arrays for:

- metadata constants
- EDD definitions
- VAR definitions
- RPTT definitions
- TBLT definitions
- CTRL definitions
- CONST definitions
- MAC definitions
- OP definitions

Where each array is named after the mnemonic for the particular AMM object, as defined in Section 5.4.1, with the exception of the metadata (MDAT) array which is unique to the ADM template itself.

In particular, the template does not provide definitions for RPT, TBL, SBR, or TBR objects as these are defined dynamically in the context of a network deployment.

The general format of the JSON ADM Template is as follows.

```
{
  "Mdat" : [],
  "Edd"  : [],
  "Var"  : [],
  "Rptt" : [],
  "Tblt" : [],
  "Ctrl" : [],
  "Const : [],
  "Mac"  : [],
  "Oper" : []
}
```

6.3. ADM Metadata

The metadata array contains CONST objects that provide information about the ADM itself.

(STR) name
: This is the human-readable name of the ADM that should appear in message logs, user-interfaces, and other human-facing applications.

(STR) namespace
: This is the Moderated Namespace of the ADM, as defined in Section 5.1.1 and string-encoded in accordance with Section 5.1.5.1.

(STR) version
: This is a string representation of the version of the ADM. ADM version representations are formated at the discretion of the publishing organization.

(STR) organization
: This is the name of the issuing organization for this ADM.

Metadata objects are encoded in the same way as CONST objects, in accordance with Section 6.6.2.

6.4. Type Encodings

This section describes the JSON encoding of AMM data types defined in Section 5.2.

6.4.1. Primitive Type Encoding

JSON data types generally have direct support for the AMM primitive data types. The mapping of AMM primitive types to JSON data types is provided in Table 3.

AMM Type	JSON Encoding
BYTE	number (0 <= # <= 256)
INT	number
UINT	number
VAST	number
UVAST	number
REAL32	number
REAL64	number
STRING	string
BOOL	boolean

Table 3: Primitive Type Encoding

6.4.2. Derived Type Encoding

In cases where an AMM derived type is simply a special interpretation of a primitive type, the JSON encoding of the derived type will be the same as the JSON encoding of the primitive type from which it derives.

6.4.2.1. Type-Name-Value

A TNV is encoded as a JSON object with three elements: "type", "name", and "value". For each item in a TNV, there are three acceptable formulations that can be used to represent the item, as

illustrated in the following table. For the examples in this table, consider the REAL32 value of PI as 3.14159.

Desc	Example
Full	{"type":"REAL32", "name":"PI", "value":3.14159}
Named Type	{"type":"REAL32", "name":"PI", "value":null}
Anonymous Type	{"type":"REAL32", "name":null, "value":null}
Anonymous Type Value	{"type":"REAL32", "name":null, "value":3.14159}
Anonymous Value	{"type":null, "name":null, "value":3.14159}

Table 4: TNV Formulations

6.4.3. Collection Encoding

The TNVC and AC collections are encoded as JSON arrays, with each object in the array represented in accordance with the JSON encoding for that object type (TNV or ARI, respectively).

An Expression is encoded as a JSON object with two elements: a type and a postfix-expr. The description of these elements is as follows.

(UINT) type
: The data type of the evaluation of the initializer expression.

(AC) postfix-expr
: A JSON array of elements where each element is an JSON encoding of an ARI in conformance to Section 6.5.

The following is an example of a JSON encoding of an EXPR object.

```
"type": "UINT",
"postfix-expr": ["Edd.item1","Edd.item2","Oper.+UINT"]
```

6.5. ARI Encoding

An ARI may be encoded as either a string or as a JSON object, with the two representations being unambiguously interchangeable. Additionally, there exists a long-form and short-form encoding of the ARI.

String encodings provide a more compact and human-readable representation of the ARI. When an ARI is represented as a string in a JSON object, it MUST be encoded in accordance with Section 5.1.5.1. If the ARI references an object that is defined in the current ADM, then the shortform string encoding may be used, as described in Section 5.1.5.1.1. The object name to be used in the string encoding is the same as the "nm" value for the JSON object encoding, as described below.

JSON object encoding of the ARI provides additional structure that makes ARI information verification easier. An ARI is encoded as a JSON object with three keys: namespace, object name, and parameters, encoded as follows.

ns

: This element identifies the namespace within which the ARI has been defined, and encoded as a string in accordance with Section 5.1.5.1. In cases where the ARI identifies an object defined in the ADM in which it is used, the ADM's namespace may be assumed as the namespace of the ADM and this element can be omitted from the ARI JSON object.

nm

: The name of an object defined in an ADM is a string defined as the concatenation of the ADMT collection defining the object, the "." separator, and the string name of the object itself. For example, an EDD defined in the Edd array and named edd1 would have the string name "Edd.edd1".

fp

: ARI formal parameters, if present, are defined as an array with each element in the array representing the JSON TNV encoding of the parameter. If a default value is not defined for the parameter, then the value of the TNV MUST be omitted.

  The fp element is not used when AMM objects are defined in the context of an ADM, as the ADM template for defining objects already includes parameter information. This element is used when AMM objects are defined in accordance with the JSON ADM syntax, but by network operators as part of network-specific configuration.

If the ARI JSON object has the fp element, then it MUST NOT have the ap element. An ARI MUST NOT define both formal and actual parameters in the same object instance.

ap

ARI actual parameters, if present, are defined as an array with each element of the array representing the JSON TNV encoding of the parameter. In cases where an optional parameter is not present, an empty TNV object will be used in its place for that parameter. The name element of the TNV MUST NOT be present for actual parameters.

In cases where the actual parameter is by value, then the TNV value key will hold the JSON encoding of the value of the parameter.

In cases where the actual parameter is by name, then the TNV MUST have the type "ParmName" and the value MUST be the string name of the parameter whose value should be used to populate the value of this actual parameter, as described in Section 5.1.3.2.

If the ARI JSON object has the fp element, then it MUST NOT have the ap element. An ARI MUST NOT define both formal and actual parameters in the same object instance.

The following are examples of JSON encoded ARI objects.

String Encoding	JSON Encoding
"N1/N2/Edd.edd1"	{"ns":"N1/N2", "nm":"Edd.edd1"}
"N1/N2/Edd.edd2(UINT num=3)"	{"ns":"N1/N2", "nm":"Edd.edd2", "fp":[{"type":"UINT", "name"="num", value":3}]}
"N1/N2/Edd.edd2()"	{"ns":"N1/N2", "nm":"Edd.edd2", "ap":[{}]}
"N1/N2/Edd.edd2(4)"	{"ns":"N1/N2", "nm":"Edd.edd2", "ap":[{"type":"UINT", "value":4}]}
"N1/N2/Edd.edd3(<input>)"	{"ns":"N1/N2", "nm":"Edd.edd3", "ap":[{"type":"ParmName", "value":"input"}]}

Table 5: Formal Parameter Encoding

6.6. ADM Structures

6.6.1. General Notes

The following guidelines apply to the JSON encoding of AMM objects.

Identification
: Objects do not include an ARI object as part of their definition. All of the contents of an ARI are derivable in the context of the ADM and adding an ARI encoding as part of the AMM object definition would be redundant and require maintaining naming information in two places in the ADM document.

Common Elements
: Every JSON encoding of an AMM object MUST have the following elements:

  * Name
    The identifier of the AMM Object. This MUST be unique across all name elements defined in the ADM collection of these types of objects.

  * Description
    A string description of the kind of data represented by this data item.

Formal Parameters
: If an AMM object may be parameterized, then an element MUST be present in the JSON object named "parmspec" which is defined as a JSON-encoded TNVC. Each element in the TNVC representing the JSON TNV encoding of the formal parameter. If a default value is not defined for the parameter, then the value of the TNV MUST be omitted.

6.6.2. Constant (CONST) Encoding

The CONST JSON object is comprised of four elements: "name", "type", "value, and "description". The description of these elements is as follows:

Name
: The identifier of the constant. This MUST be unique across all name elements for CONSTs in the ADM.

Type
: The strong typing of this data value. Types MUST be one of those defined in Section 5.4.

Value
: The value of the constant, expressed in the JSON encoding of the data type.

Description
: A string description of the kind of data represented by this data item.

The following is an example of a JSON encoding of a CONST object.

```
"name": "PI",
"type": "REAL64",
"value": 3.14159,
"description": "The value of PI."
```

6.6.3. Control (CTRL) Encoding

The CTRL JSON object is comprised of three elements: "name", "parmspec", and "description". The description of these elements is as follows:

Name
: The identifier of the control. This MUST be unique across all name elements for CTRLs in the ADM.

ParmSpec

This optional item describes parameters for this control. This is encoded as an array where each element in the array is encoded as a formal parameter in accordance with Paragraph 3.

Description
A string description of the kind of data represented by this data item.

The following is an example of a JSON encoding of an CTRL object.

```
"name": "reset_src_cnts",
"parmspec": [{"type":"STR","name":"src"}],
"description": "This control resets counts for the given source."
```

6.6.4. Externally Defined Data (EDD) Encoding

The EDD JSON object is comprised of four elements: "name", "type", "parmspec", and "description". The description of these elements is as follows:

Name
The identifier of the EDD data item. This MUST be unique across all name elements for EDDs in the ADM.

Type
The strong typing of this data value. Types MUST be one of those defined in Section 5.4.

ParmSpec
The optional array of formal parameters encoded in accordance with Paragraph 3.

Description
A string description of the kind of data represented by this data item.

The following is an example of a JSON encoding of an EDD object.

```
"name": "num_good_tx_bcb_blks_src",
"type": "UINT",
"parmspec": [{"type":"STR","name":"Src"}],
"description": "Successfully Tx BCB blocks from SRC"
```

6.6.5. Macro Encoding

The Macro JSON object is comprised of three elements: "name", "definition", and "description". The description of these elements is as follows:

Name
The identifier of the macro. This MUST be unique across all name elements for MACs in the ADM.

Definition
This is a JSON array whose elements are shorthand references are in accordance with Section 6.5 and are of the type CTRL or MAC.

Description
A string description of the kind of data represented by this data item.

The following is an example of a JSON encoding of an MAC object.

```
 "name": "user_list",
  "definition": [{
    "nm":"Ctrl.list_vars",
    "ap": []
},
{
   "nm":Ctrl.list_rptts"
   "ap": []
}],
"description": "List user defined data."
```

6.6.6. Operator (OP) Encoding

The OP JSON object is comprised of four elements: "name", "result-type", "in-type", and "description". The description of these elements is as follows.

Name
The identifier of the operator. This MUST be unique across all name elements for OPs in the ADM.

Result-Type
The numeric result of applying the operator to the series of operands. This must be one of the encodings for Table 2.

In-Type

This is an ordered JSON array of operands for the operator. Each operand is a data type encoded in accordance with Table 2.

Description
A string description of the kind of data represented by this data item.

The following is an example of a JSON encoding of an OP object.

```
"name": "plusINT",
"result-type": "INT",
"in-type": ["INT", "INT"],
"description": "Int32 addition"
```

6.6.7. Table Template (TBLT) Encoding

The TBLT JSON object is comprised of four elements: "name", "columns", and "description". The description of these elements is as follows:

Name
The identifier of the table template data item. This MUST be unique across all name elements for TBLTs in the ADM.

Columns
This is a JSON array of elements, with each element representing the definition of the type of information represented in each column. Each column is described using the same encoding as a TNV described in Table 4.

Description
A string description of the kind of data represented by this data item.

The following is an example of a JSON encoding of an TBLT object.

```
"name":"keys",
"columns": [{"type":"STR","name":"ciphersuite_names"}],
"description": "This table lists supported cipher suites."
```

6.6.8. Report Template Encoding

The RPTT JSON object is comprised of four elements: "name", "parmspec", "definition", and "description". The description of these elements is as follows:

Name

The identifier of the report template. This MUST be unique across all name elements for RPTTs in the ADM.

ParmSpec
This optional item describes parameters for this report. This is encoded as an array where each element in the array is encoded as a formal parameter in accordance with Paragraph 3.

Definition
This is an array of data elements that represent the ordered set of information associated with the report. Each element in the array is encoded as a data item shorthand in accordance with Section 6.5.
Report item elements MAY use reference parameters in their definition. In those cases, the reference parameters in the definition list MUST match report entry parameter names from the ParmSpec element in the report template definition.

Description
A string description of the kind of data represented by this data item.

The following is an example of a JSON encoding of an RPTT object.

```
{
   "name": "default_report",
   "parmspec": [{
      "type": "STR",
      "name": "endpoint_id"
   }],
   "definition": [
     {
        "ns": "DTN:bp",
                              "nm": "Edd.edd_using_a_parm",
        "ap": [{
           "type": "PARMNAME",
           "value": "endpoint_id"
        }]
     },
     {
                              "ns": "DTN:bp",
        "nm": "Edd.edd_with_default ",
        "ap": [{
           "type": "INT",
           "value": ""}
        ]},
     {  "ns": "DTN:bp",
        "nm": "Edd.edd_with_no_parms ",
        "ap": []
     }
   ]
   "description": "A default report."
}
```

6.6.9. Variables Encoding

The VAR JSON object is comprised of four elements: "name", "type", "initializer", and "description". The description of these elements is as follows:

Name
: The identifier of the variable data item. This MUST be unique across all name elements for VARs in the ADM.

Type
: The strong typing of this data value. Types MUST be one of those defined in Section 5.4.

Initializer
: The expression used to establish the initial value of the variable. This initializer is an expression encoded in conformance with Section 6.4.3.

Description
A string description of the kind of data represented by this data item.

The following is an example of a JSON encoding of an VAR object.

```
{
   "name": "total_bad_tx_blks",
   "type": "UINT",
   "initializer": {
      "type": "UINT",
      "postfix-expr": [{
         "nm": "Edd.item1",
        "ap": [{
            "type": "UINT",
            "value": 0
         }]
      }, {
         "nm":"Edd.item2",
         "ap":[{
            "type":"UINT",
            "value": 1
         ]}
      }, {
         "nm": "Oper.plusUINT",
         "ap":[]
      }]

   },
   "description": "# total items (# item1 + # item2)."
 }
```

6.6.10. Exemptions

Certain AMM objects are not intended to be statically defined in the context of an ADM document. Literals, Reports, Tables, State-Based Rules, and Time-Based Rules all only have meaning in the context of an operational network. These objects are defined by network operators as part of network-specific configuration and therefore not present in the ADM Template.

7. ADM Author Considerations

The AMM model provides multiple ways to represent certain types of data. This section provides informative guidance on how to express application management constructs efficiently when authoring an ADM document.

Use Parameters for Dynamic Information.

Parameters provide a powerful mechanism for expressing associative look-ups of EDD data. EDDs SHOULD be parameterized when the definition of the EDD is dependent upon run-time information. For example, if requesting the number of bytes through a specific endpoint, the construct num_bytes("endpoint_name") is simpler to understand and more robust to new endpoint additions than attempting to enumerate the number and name of potential endpoints when defining the ADM.

Do Not Use Parameters for Static Information.

Parameters incur bandwidth and processing costs (such as type checking) and should only be used where necessary. If an EDD object can be parameterized, but the set of parameters is known and unchanging it may be more efficient to define multiple unparameterized EDD objects instead. For example, consider a single parameterized EDD object reporting the number of bytes of data received for a specific, known set of priorities and a request to report on those bytes for the "low", "med", and "high" priorities. Below are two ways to represent these data: using parameters and not using parameters.

Parameterized EDDs	Non-Parameterized EDDs
num_bytes_by_pri(low)	num_bytes_by_low_pri
num_bytes_by_pri(med)	num_bytes_by_med_pri
num_bytes_by_pri(high)	num_bytes_by_high_pri

The use of parameters in this case only incurs the overhead of type checking, parameter encoding/decoding, and associative lookups. This situation should be avoided when deciding when to parameterize AMM objects.

Use Tables for Related Data.

In cases where multiple EDD or VAR values are likely to be evaluated together, then that information SHOULD be placed in a Table Template rather than defining multiple EDD and VAR objects. By making a Table Template, the relationships amongst various data values are preserved. Otherwise, Managers would need to remember to query multiple EDD and/or VAR objects together which is burdensome, but also results in high bandwidth and processor utilization.

8. IANA Considerations

This document defines a moderated namespace registry in Section 5.1.1.1. This registry is envisioned to be moderated by IANA. Entries in this registry are to be made through Expert Review.

This document defines a new URI scheme, "ari", as defined in Section 5.1.5.

9. Security Considerations

This document does not describe any on-the-wire encoding or other messaging syntax. It is assumed that the exchange of AMM objects between Agents and Managers occurs within the context of an appropriate network environment.

This AMM model may be extended to include the concept of Access Control Lists (ACLs) to enforce roles and responsibilities amongst Managers in the network. This access control would be implemented separately from network security mechanisms.

10. References

10.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC3339] Klyne, G. and C. Newman, "Date and Time on the Internet: Timestamps", RFC 3339, DOI 10.17487/RFC3339, July 2002, <https://www.rfc-editor.org/info/rfc3339>.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <https://www.rfc-editor.org/info/rfc3986>.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <https://www.rfc-editor.org/info/rfc4648>.

10.2. Informative References

[I-D.birrane-dtn-ama] Birrane, E., "Asynchronous Management Architecture", draft-birrane-dtn-ama-06 (work in progress), October 2017.

Authors' Addresses

Edward J. Birrane
Johns Hopkins Applied Physics Laboratory

Email: Edward.Birrane@jhuapl.edu

Evana DiPietro
Johns Hopkins Applied Physics Laboratory

Email: Evana.DiPietro@jhuapl.edu

David Linko
Johns Hopkins Applied Physics Laboratory

Email: David.Linko@jhuapl.edu