Delay-Tolerant Networking
Internet-Draft
Intended status: Standards Track
Expires: 20 July 2026

E.J. Birrane
B. Sipos
J. Ethier
JHU/APL

16 January 2026

DTNMA Application Management Model (AMM) and Data Models
draft-ietf-dtn-amm-06

Abstract

This document defines a model that captures the information necessary to asynchronously manage applications within the Delay-Tolerant Networking Management Architecture (DTNMA). This model provides a set of common managed object types, data types and structures, and a template for information needed within each application data model. The built-in definitions are made to be extensible by applications without needing to modify core Agent or Manager behavior.

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 20 July 2026.

Table of Contents

1. Introduction

The Delay-Tolerant Networking Management Architecture (DTNMA) [RFC9675] defines a concept for the open-loop control of applications (and protocols) in situations where timely, highly-available connections cannot exist among managing and managed nodes in a network. While the DTNMA provides a conceptual information model, it does not include details necessary to produce interoperable data models.

1.1. Scope

This document defines a two-level data model suitable for managing applications in accordance with the DTNMA. The two levels of model are:

1. A meta-model for the DTNMA, called the Application Management Model (AMM), which defines the object types and literal-value types used in the DTNMA in a concrete way.

2. An object model, instantiating the AMM meta-model, which is used by static Application Data Models (ADMs) and dynamic Operational Data Models (ODMs) within an Agent.

This document does not define any specific encodings of AMM values or of ADM or ODM contents. In order to communicate data models and values between DTNMA Agents and Managers in a network, they must be encoded for transmission. Specific encoding details are outside of the scope of this document, but they are discussed in Section 3.1.4 and Section 4.

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 values 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 ADM 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.

1.2. Terminology

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].

The terms "Actor", "Agent", "Manager", "Rule", "State-Based Rule", and "Time-Based Rule" are used without modification from the definitions provided in [RFC9675].

Additional terms defined in this document are the following.

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 Management Model (AMM): The object and literal-value model defined by this document in Section 3 and implemented as instances in ADMs and ODMs.

Application Resource Identifier (ARI): A unique identifier for any AMM object, namespace, or literal value defined in [I-D.ietf-dtn-ari]. The text form of an ARI is conformant to the Uniform Resource Identifier (URI) syntax documented in [RFC3986] and using the scheme name "ari".

Application Data Model (ADM): The set of statically-defined objects necessary to manage an application asynchronously. This is also the name for the specific syntax used to express the contents of that ADM, as defined in another document.

Operational Data Model (ODM): The set of dynamically-defined objects created and controlled by Managers in the network. There is currently no specific syntax used to express the entire contents of an ODM outside of data from introspection reports generated by an Agent.

Namespace: Each ADM and ODM has a universally unique identifier and acts as a namespace for a set of AMM objects. A namespace is identified by its organization ID, model ID, and (for ADMs) a specific model revision. A namespace reference is also a form of AMM value (Section 3.1.3) as realized in ARI syntax.

Identity Object (IDENT): An object type used as an extensible form of enumerated value as defined in Section 3.4.3. These objects have no specific state within an Agent and are characterized entirely by their identity.

Built-In Type: The set of value types which form the baseline of the AMM typing system as defined in Section 3.2. These include sized integer types, text and byte strings in Section 3.2.1, more complex nested containers in Section 3.2.2 and object references in Section 3.2.3.

Semantic Type: This is a means of combining, constraining, or annotating existing built-in types or TYPEDEF types as defined in Section 3.3. A typical expected use of semantic type is to annotate the use of an integer counter type with an engineering unit. A semantic type on its own does not have an intrinsic identifier, but can be used in a TYPEDEF to give it identifiers.

Type Definition (TYPEDEF): An object type used to associate a name and enumeration, in a specific object namespace, with a semantic type as defined in Section 3.4.2. These objects have no specific state within an Agent and are characterized entirely by their identity and semantic type.

Value Production: An activity performed within an Agent as defined in Section 6.5. The target of a production can be an object reference (Section 3.1.2) to any value-producing object (Section 2.1). A convenience semantic type is defined for valid production targets in Section 4.2.5.

Constant (CONST): An object type used for Value Production and defined in Section 3.4.5. These objects have state maintained by an Agent. Constant objects can be used to store macro values, report template values, threshold values for rule expressions, _etc._

Variable (VAR): An object type used for Value Production and defined in Section 3.4.10. These objects have state maintained by an Agent.

Externally Defined Data (EDD): An object type used for Value Production and defined in Section 3.4.4. These objects have state obtained by an Agent from an Application.

Execution: An activity performed within an Agent as defined in Section 6.6 as directed by a Manager or as the action of a triggered Rule. A Manager can trigger execution by sending an Execution-Set (EXECSET) values to an Agent through some transport binding. A convenience semantic type is defined for valid execution targets in Section 4.2.4.4.

Control (CTRL): An object type used for Execution and defined in

Section 3.4.6. These objects have execution behavior triggered by an Agent into an Application. A set of general purpose built-in controls is defined in the Agent ADM (Section 4.3).

Macro (MAC): A semantic type for values used during Execution as defined in Section 4.2.4.2. Macros are used to control ordering of execution items and limit execution failures (an item of a macro which fails to execute will skip any following items). Macros are expected to be stored in CONST objects but can be present in any value-producing object (Section 2.1).

Agent State: The Agent state is the aggregation of all of the states of its Section 3.4.6 and Section 3.4.10 objects (within both ADMs and ODMs) and the hidden states which would be produced by Section 3.4.4 objects. These states can be used with Evaluation to trigger Section 3.4.8 actions.

Ephemeral values (such as EXECSET values waiting to be executed or RPTSET values waiting to be sent) are not considered part of this definition of the agent state. However, such ephemeral information can be reflected by EDDs in the Agent ADM (Section 4.3) that count or list Agent queue contents.

Agent Timeline: An Agent timeline is used for internal bookkeeping of future events. One purpose of timeline events is to trigger actions for enabled Section 3.4.9 objects. Another purpose is to support built-in "waiting" controls in the Agent ADM (Section 4.3).

Evaluation: An activity performed within an Agent as defined in Section 6.7. The target of an evaluation can be the predicate of a State-Based Rule (SBR) object used for autonomy, or an item within a Report Template (RPTT) value during Reporting. A convenience semantic type is defined for valid evaluation targets in Section 4.2.4.5.

Operator (OPER): An object type used for Evaluation and defined in Section 3.4.6. These objects have evaluation behavior triggered by an Agent into an Application. A set of general purpose built-in operators is defined in the Agent ADM (Section 4.3).

Expression (EXPR): A semantic type for values used during Evaluation as defined in Section 4.2.4.1. Expressions are lists of operators and operands in postfix order which are evaluated in a specific context on an Agent. Expressions are expected to be stored as TBR predicates and in CONST objects but can be present in any value-producing object (Section 2.1).

Reporting: An activity performed within an Agent as defined in Section 6.8. The target of reporting is always a Report Template (RPTT) containing items to evaluate into one report of a Reporting-Set (RPTSET). The trigger for reporting is either a built-in control in the Agent ADM (Section 4.3) or the completion of execution of an Execution-Set (EXECSET) target when a nonce value is included.

Report Template (RPTT): A semantic type for values used during Reporting as defined in Section 4.2.4.3. Report templates are lists of items used as a schema for report contents. Report templates are expected to be stored in CONST objects but can be present in any value-producing object (Section 2.1).

Report: A report is the output of the Reporting activity and kept within Reporting-Set (RPTSET) values. An Agent sends its reports to a specific Manager through some transport binding.

2. Data Modeling Concept of Operations

In order to asynchronously manage an application in accordance with the [RFC9675], an application-specific data model must be created containing static structure 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. ADM structure and base ADMs are discussed in detail in Section 4.

The objects codified 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 objects. Each namespace of dynamic objects is termed an Operational Data Model (ODM) and is discussed in detail in Section 5.

Both the ADMs and ODMs rely on a common meta-model, the Application Management Model of Section 3, which defines the basic structure of what kinds of types and objects are available to use within the DTNMA. The relationships among the AMM, ADM, and ODM are illustrated in Figure 1. Together, the set of objects in the union of all supported ADMs with dynamic ODM objects forms the data model used to manage an Agent.

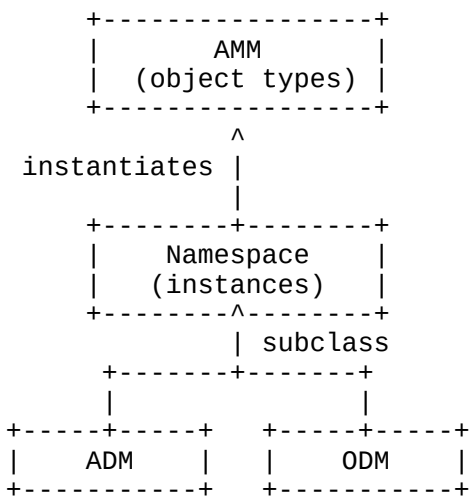

Figure 1: Data Model Relationships

The AMM defines a strict separation between long-lived object instances and ephemeral value instances. While an Agent hosts the object instances, each manager must contain the corresponding ADM and ODM definitions in order to identify and interact with those objects. Those interactions are performed using Application Resource Identifiers (ARIs) as depicted in Figure 2.

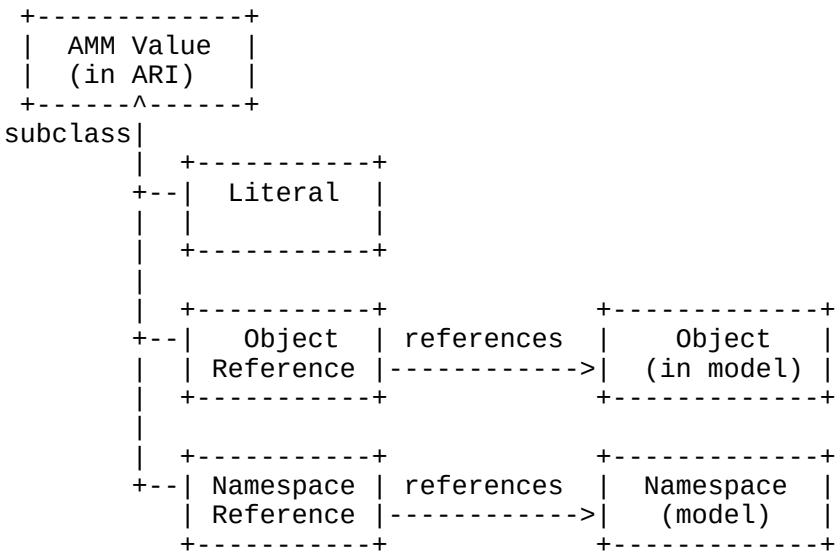

Figure 2: AMM Value and Object Relationships

While an agent hosts the actual object instances, each manager must contain the corresponding ADM and ODM definitions in order to identify and interact with those objects. Those interactions are performed using Application Resource Identifiers (ARIs) as depicted in Figure 3.

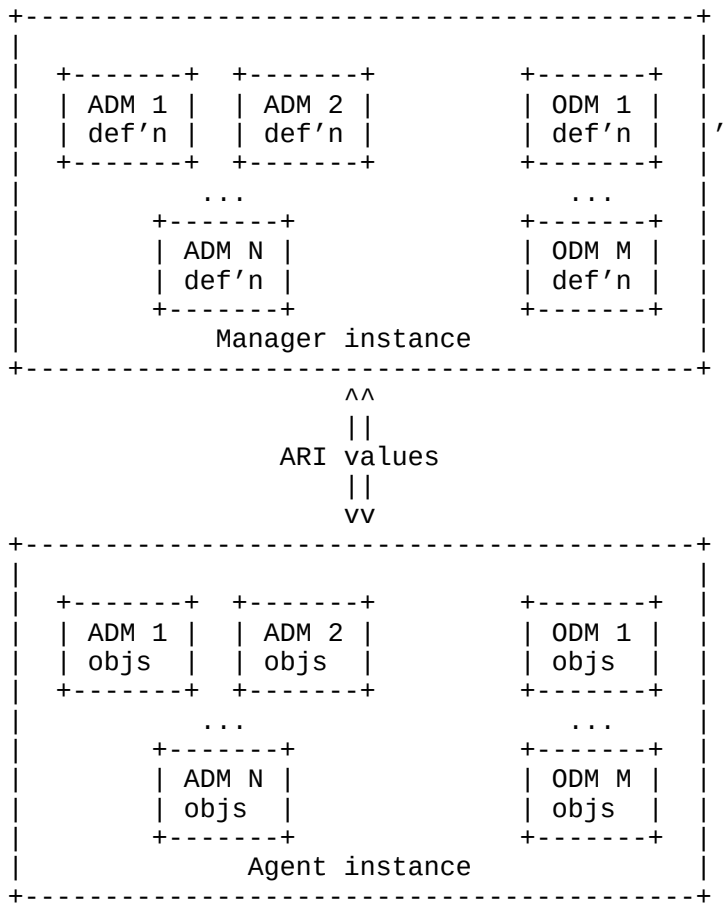

Figure 3: Agent and Manager Interaction

2.1. Values and Value-Producing Objects

The logical structure of AMM values is defined in Section 3.1, which is used as the basis for Agent Processing activities and for the basis of Agent-Manager Messaging contents. Values are realized and encoded according to a separate ARI specification of [I-D.ietf-dtn-ari].

Of the value-producing object types discussed in Section 3.4 and Section 6.5, the functions of these objects are summarized and compared with literals in Table 1 and the following list.

Literal Values: ARI literals are, by definition, immutable and fully self-contained values.

For example, the number 4 is a literal value. The name "4" and the value 4 represent the same thing and are inseparable. Literal values cannot change ("4" could not be used to mean 5) and they are defined external to the autonomy model (the autonomy model is not expected to redefine what 4 means).

Constant (CONST): These objects are named values which are defined in ADMs or ODMs and produced directly by the Agent implementing the model. The name, data type, and value of the constant are fixed and cannot be changed during its lifetime (see Section 7.1).

An example of a constant would be defining the numerical value _pi_ to some predetermined precision. Another typical example of a constant is the definition of a report template (Section 4.2.4.3) or macro (Section 4.2.4.2).

Variable (VAR): These objects are named value storage entities which are defined in ADMs or ODMs and managed by the Agent implementing the model. While the name and data type constant, the value can change over time due to controls acting upon the Agent or changes from the application itself. One standard interface is an ADM-defined initial state value with a control available to reset to that initial state. Another standard interface is a control to set a variable to a specific value, based upon evaluating an expression within the Agent at runtime.

An example of a manager-controlled variable would be a threshold value used to compare against a sensor value in a rule predicate. Another example is a variable which configures some algorithm within the associated application, and changes to the variable state are reflected in changes within the application.

Externally Defined Data (EDD): These objects are named entities which are defined in an ADM but produce values based on data provided to an Agent from its environment. These values are the foundation of state-based autonomy as they capture the state of the managed device. The autonomy model treats these values as read-only state. It is an implementation matter to determine how external data is transformed into values of the specific type specified for an EDD.

Examples of externally defined values include temperature sensor readings, the instantaneous data rate from a modem, or introspection on the configured state of an application.

Within this comparison table, "internal" means values are managed by the Agent itself and "external" means the source of values is outside the Agent.

	Immutable	Mutable
Internal	CONST	VAR
External	_Literal_	EDD

Table 1: Value-Producing Object Types

2.2. Agent Processing

Based on the reasoning described in [RFC9675], much of the closed-loop processing of the state of the DTNMA Agent is performed on the Agent itself using rule objects. The different types of processing performed on the Agent are separated into Execution, Evaluation, and Reporting with corresponding AMM object types related to each of these as indicated in Table 2 (_e.g._, execution relates to CTRL objects but not OPER objects). Some of the objects defined in the Agent ADM (Section 4.3) combine the use of these processing activities, but they are still independent of each other. There is no mixing of activities such as executing a control within an expression; although the execution of a control can result in an expression being evaluated they are independent activities.

Within the runtime of an Agent any input, output, and intermediate values can use the concept of a semantic type (Section 3.3) to do things like restrict the valid numeric range of a value or allow a union of disjoint types to be present (_e.g._, a certain value can be a boolean or an unsigned integer). This is combined with the built-in types available (see Section 3.2) to allow complex type information to be present in an ADM or ODM without requiring additional over-the-wire encoding size. A Type Conversion activity is defined for when implicit or explicit type conversion is needed.

Activity	Objects	Values
Execution	CTRL	MAC, EXECSET
Evaluation	OPER, TYPEDEF	EXPR
Reporting	_N/A_	RPTT, RPTSET
Value Production	CONST, EDD, VAR	_N/A_
Type Matching and Conversion	TYPEDEF	ARITYPE
Rule Autonomy	SBR, TBR	_N/A_

Table 2: Processing Activities and Object Types

2.3. Agent-Manager Messaging

This document does not define a messaging protocol between agents and managers but full functioning of this data model behavior does rely on the following types of messages being available. Because each message is based on ARI value types, they can be implemented in Agent and Manager by encoding the associated ARI according to a network-specific transport binding.

The choice of encoding form, framing, or transport are implementation matters outside of this specific document. A specific message framing and an initial binding to the Bundle Protocol are defined in [I-D.ietf-dtn-amp].

Execution: This message causes an Execution of a referenced parameterized CTRL object (Section 3.4.6) or MAC value (Section 4.2.4.2). The form of this message is an Execution-Set (EXECSET) value. This type of message is only sent from Manager to Agent. Each message can contain multiple execution targets but all must be associated with the same nonce value. It is an implementation detail whether a Manager sends fewer messages with more targets or more timely messages with fewer targets.

Reporting: This message carries the reports generated by Reporting

activities and as the result of Execution when the Manager provides a correlator nonce. The form of this message is an Reporting-Set (RPTSET) value. This type of message is only sent from Agent to Manager. Each message can contain multiple report containers but all must be associated with the same nonce value. It is an implementation detail whether an Agent sends fewer messages with more reports or more timely messages with fewer reports.

In addition to the ARI values being conveyed in those messages, transport bindings need to include endpoint identity information for each message. Each transport binding SHALL convey a destination endpoint identity with each transmitted message. Each transport binding SHALL ensure that transmitted messages are associated with an authenticated source endpoint identity. Each transport binding SHALL require that received messages are associated with an authenticated source endpoint identity. Each transport binding SHALL convey an authenticated source endpoint identity with each received message. The form of an endpoint identity and means by which an identity is authenticated are transport-specific and outside the scope of this document.

In addition to the data messaging above, if a transport binding provides the Agent or Manager with information about the reachability of its peer Managers or Agents, respectively, the entity SHOULD use that information to queue (and aggregate) messaging to unreachable peers.

| For DTN transports, as expected to be used in [RFC9675], true destination reachability is not knowable at network edges and this capability is not expected to be useful. For resource-constrained applications of the DTNMA, reachability information is expected to be more available and useful for queuing. For example, if all reporting from an agent occurs through a single outgoing data link and that link becomes unavailable then any managers are certainly unreachable.

3. Application Management Model (AMM)

This section describes the Application Management Model, which is the meta-model used to implement the DTNMA. This section also provides additional information necessary to work with this model, such as: literal value types, object structure, naming conventions, and processing semantics.

The overall AMM is decomposed into two categories:

Objects: These are the structural and behavioral elements of the

AMM, present in ADMs and ODMs. Objects implement the actual purpose of the applications being managed; they extract values from the Agent's environment, operate on expressions, store variables, and execute controls to affect the Agent's environment. Because objects are behavioral they have no complete static representation, objects are only ever described and identified. AMM object types are defined in Section 3.4 and objects are instantiated as part of an ADM or ODM.

Values: These are the runtime state and intermediates of the AMM, present in the Agent's state but not directly in ADMs or ODMs. Objects produce, operate on, and store or consume values and these values are what are present in messaging between Managers and Agents. AMM values are explained in more detail in Section 3.1. One subset of AMM values are object references used to identify (and parameterize) individual AMM objects.

3.1. AMM Values

Values within the AMM have two top-level classes: literal values, and object reference values. Each of these is discussed more detail in the following subsections. Both classes of AMM values are related to what can be represented externally as an ARI, as described in Section 3.1.4.

3.1.1. Literal Values

Literal values are those whose value and identifier are equivalent. These are the most simple values in the AMM. For example, the literal "4" serves as both an identifier and a value.

Because the Literal value serves as its own identifier, there is no concept of a parent namespace or parameters. A literal can be completely identified by its data type and data value.

Literals have two layers of typing:

Built-In Type: This is the lower layer which defines the syntax of the literal value and bounds the domain of the value (_e.g._ a BOOL has two possible values, while an INT has a large domain of integers). There are a small number of built-in types as described in Section 3.2 and they are managed with an IANA registry defined in Section 9.3 of [I-D.ietf-dtn-ari].

Semantic Type: This is the higher layer which defines additional semantics to a value, such as a restricted domain of values or a human-friendly text unit or limits on the items of an ARI collection. Semantic types are defined within ADMs (see

Section 3.3 and Section 3.4.2), so there can be an arbitrary number of them and they are managed outside of a central authority.

All literal values have a concrete and stand-alone representation independent of any ADM or ODM behavior in the form of an ARI (Section 3.1.4) and a value itself has no direct association with a semantic type outside of a specific context in which that type is used (_e.g._, Section 6.10 and Section 6.11).

3.1.2. Object Reference Values

Every object in the AMM is uniquely identifiable, regardless of whether the item is defined statically in an ADM or dynamically in an ODM. Object reference values are composed of the following parts: an organization ID, a model ID, an optional model revision date, an object type, an object ID, and object-specific optional parameters.

The organization ID, model ID, and model revision together identify a single object namespace as discussed in Section 3.1.3.

A registry for organization text names and integer enumerations is defined in Section 9.3 of [I-D.ietf-dtn-ari]. This registry makes a reservation for a large number of private-use code points to allow operating domains, missions, and users to define their own non-registered organization space for ADMs and ODMs.

Object types, each with a text name and integer enumeration, are defined in an IANA registry by Section 9.3 of [I-D.ietf-dtn-ari].

Object names are text strings and enumerations whose value is determined by the creator of the object. For those objects defined in an ADM, the structure of the object name is given in Section 3.4.1.

3.1.2.1. 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 type is defined in Section 3.4 with behaviors related to parameters defined in Section 6.

There are three notions of parameters defined in the AMM, which take their name from computer programming vernacular used for discussing function declarations and function calls, those are: formal parameters, given parameters, and actual parameters. Formal parameters are discussed in Section 3.4.1 while given and actual parameters are discussed here in relation to the object reference.

Given parameters are part of an object reference and represent the data values passed to a parameterized AMM object at runtime. They "fulfill" the parameter requirements defined by the formal parameters for that object. Each object type can have a slightly different notion of how its parameters affect its processing activities.

Given parameters can either be absent, present as a list (with logic equivalent to the built-in AC type), or present as a map (with logic equivalent to the built-in AM type). Either form can be used to convert to equivalent actual parameters as discussed below.

There are two ways in which the value of each given parameter can be used:

Parameter by Value: This method involves directly supplying the value as part of the given parameter. It is the most direct method for supplying values.

Parameter by Label: This method involves supplying the LABEL of some other processing-context-specific value and substituting, at runtime, that named value as the value of this parameter. This method is useful when a parameterized AMM Object produces a value that references a parameter of the producing object. The produced value’s given parameter can be given as the LABEL of the producing object’s formal parameter. In this way, a value-producing object’s parameters can "flow down" to all of the values that it produces.

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

Finally, actual parameters are the result of applying the Parameter Handling procedure to normalize a set of given parameters based on a set of formal parameters from a processing context.

3.1.3. Namespace Reference Values

One form of AMM value is a namespace reference, which is similar to an object reference but only includes the namespace components of the organization ID, model ID, and model revision .

AMM objects are identified 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.

Namespaces, by their existence, SHALL NOT be used as a security mechanism. An Agent or Manager SHALL NOT infer security information or access control based solely on namespace information.

3.1.4. The Application Resource Identifier (ARI)

The Application Resource Identifier (ARI) is used to represent AMM values outside of an Agent or Manager (_i.e._ in messaging between them) and is defined in [I-D.ietf-dtn-ari]. Another function of the ARI is for diagnostic or configuration purposes within Managers or Agents.

3.2. Built-In Value Types

This section describes the built-in types used for AMM values, which are those usable directly with ARI syntax. By definition, literal values are self-contained and literal types restrict the form and function of those values.

All built-in types within the AMM exit within a flat namespace, but some types have complex relationships with other types beyond the "is a" concept of type inheritance. Built-in types are defined within the "DTNMA Literal Types" and "DTNMA Object Types" registries of [IANA-DTNMA] and explained in this section. The following subsections divide the types into groups to simplify their explanation, not because of an intrinsic relationship within each group.

These lists of built-in type names are not fixed in any single specification, and require standards action to add to and update URI processors to handle them, so it is expected that this list will be relatively static (compared to the expected rate of addition or changes to ADMs).

3.2.1. Simple Types

Simple types are those which cannot be subdivided and represent an "atomic" value within the AMM type system. They correspond roughly with the CBOR primitive types Section 3.3 of [RFC8610]. The simple types are summarized in Table 3.

Type	Description
NULL	The singleton null value.
BOOL	A native boolean true or false value.
BYTE	An 8-bit unsigned integer.
INT	A 32-bit signed integer.
UINT	A 32-bit unsigned integer.
VAST	A 64-bit signed integer.
UVAST	A 64-bit unsigned integer.
REAL32	A 32-bit [IEEE.754-2019] floating point number.
REAL64	A 64-bit [IEEE.754-2019] floating point number.
TEXTSTR	A text string composed of (unicode) characters.
BYTESTR	A byte string composed of 8-bit values.
TP	An absolute time point (TP).
TD	A relative time difference (TD) with a sign.
LABEL	The text name or integer enumeration of a non-object AMM entity in a specific context (_e.g._, formal parameter name/index or table column name/index). This is only valid in a nested parameterized ARI or for specific control parameters.
CBOR	A byte string containing a single encoded CBOR item. The structure is opaque to the Agent but guaranteed well-formed for the ADM using it.
ARITYPE	The text name or integer enumeration of one of the built-in value types (as registered in

	[IANA-DTNMA]).	
OBJPAT	A pattern for matching AMM objects using ranges within their hierarchy of identifiers. These are distinct from object reference (Section 3.2.3) types, do not need to correlate to specific objects, and can never convey parameters.	

Table 3: Simple Literal Types

The following subsections discuss nuances in sub-groups of these simple types.

3.2.1.1. Discrete Value Types

The NULL and BOOL types are used to limit to specific discrete values. Because there are CBOR primitive types corresponding exactly with these AMM types, generators of ARIs with these types can always be compacted by eliding the literal type as defined in Section 6.13.

The NULL type has only a single value, null, which is not useful for expressions or type casting but is useful for defining union types which have "optional value" semantics where the null value is used to indicate the absence of a normal value.

The BOOL type is useful for type casting (Section 6.11.1) where an arbitrary value is treated as "truthy" or "falsey" in a context such as a State-Based Rule (SBR) condition.

3.2.1.2. Numeric Types

All of the numeric types (BYTE, UINT, INT, UVAST, VAST, REAL32, and REAL64) exist within a domain where values can be converted between types (Section 6.11.2) if the values fit within the numeric domain of the destination type. Some cases of implicit casting is done for type promotion as necessary for arithmetic operations.

3.2.1.3. Absolute and Relative Time Types

The absolute time point (TP) type represents an instant in time in the UTC datum, consistent with the full date-and-time representation of internet timestamps [RFC3339].

The relative time difference (TD) type represents an offset in time from a relative epoch instant, either later than (a positive offset) or earlier than (a negative offset). This type is consistent with a signed time duration defined for internet timestamps [RFC3339]. The epoch instant of a relative time needs to be unambiguously defined in the context using the time value.

| In cases where a type signature contains an union of TP and TD (i.e. an option for either type), relative times have some 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.

3.2.1.4. Object Reference Pattern Type

The object reference pattern (OBJPAT) type represents a choice from one or many possible identifier values for each path segment of an object reference value (Section 3.1.2) path. One extreme of the capability of a single pattern is the ability to match exactly one object by its unique path (with each pattern segment is either a text name or integer enumeration). The other extreme is to have a single canonical pattern to be able to match all possible objects with wildcards (including future objects which do not exist at the time the pattern is created).

Similarly to the object reference values, these patterns contain four separate parts (organization ID, model ID, object type, and object ID) and are used to match the four parts of an object path independently. These types of pattern are intended to be implemented as part of an Agent, so there is no ability to discriminate on the model revision, if present, for a namespace.

Each part of the pattern contains one of the following to match the corresponding part of the object path:

Single value: This will match only a single identifier value as either integer enumeration or text name.

Range: This will match any identifier value contained in a disjoint set of integer intervals.

Wildcard: This will match any possible identifier in that part.

The activity for determining whether or not an object matches a specific pattern is defined in Section 6.4. This is a distinct activity from dereferencing a single object reference in Section 6.3.

3.2.2. Containers

AMM objects, or parameters associated with those objects, often need to represent groups of related data or more complex nesting of data. These are the literal types for AMM value containers, which can only be present in a typed-literal ARI form.

The AMM defines three collection literal types (AC, AM, and TBL) and allows ADMs to combine these built-in literal types with a constraint logic to create semantic types for their contents (_e.g._, for macros and expressions in Section 4.2.4.1).

3.2.2.1. ARI Collection (AC)

An ARI Collection (AC) is an ordered list of ARI elements. The contents of an AC can be restricted in size and type by the use of a semantic type (Section 3.3).

An AC is used when there is a need to refer to multiple AMM values as a single unit. For example, when defining a Report Template, the definition has an AC that defines the ordered ARIs whose values constitute that report.

3.2.2.2. ARI Map (AM)

An ARI Map (AM) is a mapping from a set of "key" ARIs to arbitrary-typed "value" ARIs. AM keys are limited to untyped literals, while the AM values can be any type. The contents of an AM can be restricted in size and type by the use of a semantic type (Section 3.3).

An AM is used when there is a need to define data structures with complex, optionally present attributes. For example, as control parameters used to define new objects in an ODM.

3.2.2.3. ARI Table (TBL)

An ARI Table (TBL) is a collection of values which are logically structured as a two dimensional table of rows and columns, with each cell of the table containing an AMM value.

Although the contents of a TBL can be handled independently of any data model, the meaning of a TBL can only be interpreted within the context of a Table Template (TBLT) defined within an ADM. The TBLT

takes the form of a structured type definition on a value-producing object which defines the columns of the table, including each of their column names and types and optional constraints on the number of and uniqueness of rows in the TBL.

A TBL is used when an EDD represents a set or list of complex items as rows in a table. For example, the Agent ADM reports its own set of supported ADMs and features as a TBL (see the "capability" object).

3.2.2.4. Execution-Set (EXECSET)

An Execution-Set (EXECSET) is a collection of values used as targets for the Execution activity. Each message can reference multiple execution sources (CTRL and MAC) and, unlike the MAC execution itself, can be handled by executing multiple items in parallel.

The contents of an EXECSET value are as follows:

Correlator nonce:
: This field is an optional opaque correlator "nonce" which can be used to indicate that the result of the corresponding CTRL execution is desired to be reported back to the Manager. The value is limited to match the NONCE (Section 4.2.5) type.

Targets:
: This is an unordered list of targets to be executed by an Agent. Each execution target is limited to match the exec-tgt (Section 4.2.4.4) type.

3.2.2.5. Reporting-Set (RPTSET)

A Reporting-Set (RPTSET) is a collection of report containers, where each report container consists of a timestamp and an ordered list of data values populated in conformance to a source object being reported on. Reporting-Set values and reports themselves do not have individual identifiers, rather they are identified by their source and the timestamp at which their data values were collected.

The contents of an RPTSET value are as follows:

Correlator nonce:
: This field is an optional opaque correlator "nonce" which is used to associate report containers with specific EXECSET messages which caused the reports to be generated. The value is limited to match the NONCE type (Section 4.2.5).

Reference time:
: This field is used as an absolute reference time for all reports contained in the RPTSET. The value is limited to match the TP built-in type. It is used as an storage optimization when a large number of reports are generated around the same time.

Report list:
: The main content of the RPTSET are the reports themselves, which are defined below. The order of reports within the RPTSET are not significant, and the presence of a report in any particular RPTSET is not significant. The RPTSET itself is only a container.

The contents of each report within a RPTSET are as follows:

Source:
: The source of the report in the form of an ARI with an object-reference for one of the following types: VALUE-OBJ (Section 4.2.5), or CTRL. If the source was parameterized, this ARI SHALL contain the actual parameters used at the time of reporting.

Generation Time:
: The timestamp at which the report items were sampled, relative to the Reference Time of the containing RPTSET. The value is limited to match the TD built-in type.

Items:
: A list of values corresponding to the source object, with cardinality according to the following:

  * For a VALUE-OBJ source the item list SHALL be the result of reporting (Section 6.8.1) on that object.

  * For a CTRL-REF source there SHALL be a single value representing the Result of the execution. A result of undefined indicates a failure executing the CTRL.

3.2.3. Object Reference Types

For each of the AMM Object Types there is a corresponding object reference type. The object type names and enumerations from the "DTNMA Object Types" registry of [IANA-DTNMA] are used as names for the built-in type of the corresponding object reference.

For example, a reference value for a CTRL object is typed as CTRL. It is important to understand the distinction, illustrated in Figure 2, between the built-in type and the object type both of which have the same name but used in two completely independent contexts.

3.2.4. Value-Class Types

As a special case of built-in type which act as a union or class of types are those listed in Table 4. These are implemented as a built-in type rather than a semantic type because they behave differently than a Type Union (Section 3.3.6) because they will match any value in the associated class and conversions within these types will not affect the value.

Type	Description
LITERAL	Any possible literal value.
OBJECT	Any possible object reference value.
NAMESPACE	Any possible namespace reference value.

Table 4: Value-Class Types

3.2.5. Custom Types

When an application requires a more complex or specialized literal type than one already available the preferred design procedure is as follows:

1. If an existing ADM already defines a semantic typedef (see Section 3.4.2) it is RECOMMENDED to import that ADM and use its typedef.

2. Otherwise, when it is possible to use an ADM-defined semantic typedef to achieve the desired goals it is RECOMMENDED to do so.

3. Otherwise, when the desired behavior cannot be accomplished by a semantic typedef, it is RECOMMENDED to use the opaque CBOR type with interface documentation to explain the syntax of the encoded CBOR item.

4. Otherwise, the application MAY make use of the private-use block of literal type code points.

Implementing a custom literal type requires implementation effort on both an Agent and its associated Manager(s) as well as being more opaque to diagnostic tools and middleboxes.

3.3. Semantic Value Types

While built-in types control the basic syntax and domain of AMM values, the concept of semantic type is to provide a means to augment literal types by expanding (via union), narrowing (via constraints), and adding human-friendly annotation (such as references to defining documents, or explanations of purpose).

Semantic types can be defined in two ways: a named Semantic Type Definition (TYPEDEF) or an anonymous semantic type defined at the point of use (_e.g._, within an AMM object definition). The specific syntax used to define semantic types within an ADM are defined and explained in a separate document.

When a "type" is needed for an AMM value in an object definition it SHALL be either one of the built-in types, a namespace-qualified semantic type, or an anonymous semantic type just for that value.

The actual mechanics of semantic typing are based on the classes defined in the following subsections.

3.3.1. Named Type Use

The simplest case is where an existing named type is referenced to be used in a specific context. The form of a named type use SHALL be an AMM value containing either an ARITYPE literal, for a built-in type (Section 3.2), or a TYPEDEF object reference, for a data-model-provided semantic type (Section 3.4.2).

Even an unconstrained type reference can be used within a TYPEDEF to provide a human-friendly name or associated documentation for the use of a simple type. While the tooling might not care about direct type use, it can greatly improve human interpretation of a data model.

An implementation SHALL be able to handle situations where type references create a loop. This would allow values to follow a recursive structure, but it does not mean the values themselves would be of an indefinite size.

Within a named type use, annotations can be added which enhance human understanding of the type. Annotations possible within a named type use SHALL consist of:

* A free-form text reference to a specific document defining the type in more detail.

* A free-form text description of the type, which can be in addition to a reference.

* A units name for NUMERIC values to make the interpretation of values more explicit.

* A display hint to enable type-specific handling of values, such as IP addresses within BYTESTR values.

Within a named type use, constraints can be added to some of the Simple Types in order to limit what values are considered valid within the type domain. Constraints possible within a named type use SHALL consist of:

* Limits on ranges of valid NUMERIC types

* Limits on length of TEXTSTR, BYTESTR, or CBOR

* Human-friendly names of enumerated values or bit positions for INTEGER types

* Regular expression patterns for TEXTSTR

3.3.2. Uniform List

This is the case of a list of AMM values within an ARI Collection (AC) where the type of each value is uniform for the whole list. Only the AC type MAY be refined by a uniform list. Each item of a uniform list SHALL be constrained to a single semantic or built-in type. The number of items in the list MAY be constrained within a range of valid sizes.

3.3.3. Diverse List

This is the case of a list of AMM values within an ARI Collection (AC) where the type of each value is different throughout the list. Only the AC type MAY be refined by a diverse list. Each part of a uniform list SHALL be constrained as either: an item with a single semantic or built-in type or a Sequence (Section 3.3.7).

3.3.4. Uniform Map

This is the case of a map of AMM values within an ARI Map (AM) where the type of each key and each value is uniform for the whole map. Only the AM type MAY be refined by a uniform map use.

3.3.5. Table Template

This is the case of a table of AMM values within a ARI Table (TBL) where each column is annotated with a text name and the type of each value in a column is uniform across all rows. Only the TBL type MAY be refined by a table template. The number of rows in the table MAY be constrained within a range of valid sizes. A single "key" column SHOULD be identified as the unique identifier for each row. One or more column tuples MAY be identified as unique among all rows.

3.3.6. Type Union

This creates a choice between a combination of multiple semantic types. Each of the types in a union SHOULD be exclusive to avoid ambiguity in interpretation by a value processor. The order of choices within a union SHALL be used as the order to check for Type Matching and Type Conversion procedures.

3.3.7. Sequence

This creates a subset of a Diverse List (Section 3.3.3) which matches multiple sequential elements of the list. A sequence is similar to a Uniform List except that it doesn’t specify an AC container, it is used to specify items _within_ a container. Each item of a sequence SHALL be constrained to a single semantic or built-in type. The number of items in the sequence MAY be constrained within a range of valid sizes.

A sequence can also be used with formal parameters to create a form of variadic parameter, where multiple given parameters are matched and combined into a single actual parameter (see Section 6.3.1).

3.4. AMM Object Types

This section identifies the types of objects that make up the AMM and which are instantiated within each ADM and ODM. Each object type is defined by its logical structure and its behavior in Value Production, Execution, or Evaluation contexts within Agents. Each type can allow or disallow parameters within objects and, due to processing behaviors, can either allow or disallow use within an ADM or ODM.

The names for the types of objects defined in this section can be used in two different and separate contexts: as a name for the type of the object itself (written as plain text within this document) when in the context of the AMM object model, or as the name of an object reference (Section 3.1.2) type (written in typewriter text within this document) when used in the context of the AMM value model.

Unless explicitly specified in the object type subsection, an object SHALL NOT be parameterized.

3.4.1. Common Object Fields

Every object type in the AMM includes a set of fields 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.

The metadata supported by the AMM for all objects is as follows:

Name:
: An object name is a text 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 with the text form of an ARI. Each object definition SHALL contain a name field. An object's name SHALL NOT change between ADM revisions. Each name SHALL conform to the id-text ABNF rule below. Within each namespace and object type, the name of an object SHALL be unique.

```
id-text = (ALPHA / "_") *(ALPHA / DIGIT / "_" / "-" / ".")
```

Enumeration:
: An object enumeration is an integer associated with the object, which identifies the object just like its name. Object enumerations provide a stable and concise identifier for the binary encoded form of an ARI. Each object definition SHOULD contain an enumeration field. An object's enumeration SHALL NOT change between ADM revisions. When present, each enumeration SHALL be a non-negative value in the domain of a signed 32-bit integer value. Within each namespace and object type, the enumeration of an object SHALL be unique.

Status:
: Each object definition MAY contain a status field. The valid status values of an object are the same as the valid status values for an ADM in Section 4.1.1. In the absence of a status field, the status of the object SHALL be considered the same as the status of the ADM which contains it.

Reference:
: Each object definition MAY contain a reference field. A reference is a text string referring to a specification or other document which details the source or purpose of the object.

Description:
: Each object definition MAY contain a description field. A description is a text string explaining the purpose or usage of the object in a human-readable format. There is no minimum or maximum size of description text for an object. 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.

Formal parameters define a method to customize an AMM object. When used by an object definition, it’s formal Parameters SHALL be an ordered list of individual formal parameter definitions. Each formal parameter SHALL include type and name. Each formal parameter MAY include an optional default value. The application of default parameters and relationship of actual parameters (Section 3.1.2.1) to formal parameters is defined in Section 6.3.1.

3.4.2. Semantic Type Definition (TYPEDEF)

An ADM can define a semantic type definition (TYPEDEF) to give a name to a semantic type (Section 3.3). This TYPEDEF name can then be used as a type anywhere else in the same ADM or another one which imports it.

The definition of a TYPEDEF consists of the following:

Name, Enumeration, Status, Reference, Description:
: As defined in Common Object Fields.

Type:
A TYPEDEF definition SHALL include the type being named, as described in Section 3.3. The type of a TYPEDEF is fixed and SHALL NOT change between ADM revisions. The type SHALL be either a union of other types or a restriction of or annotation upon another type.

As defined in this document, TYPEDEFs and semantic types can only be defined within an ADM. Future capability could allow the use of TYPEDEFs within ODMs.

3.4.3. Identity Object (IDENT)

An ADM can use an identity object (IDENT) to define a unique, abstract, and untyped identity. The only purpose of an IDENT is to denote its name, parameters, and existence semantics. This allows an extensible but controlled enumeration of valid AMM values for object parameters and states (_e.g._, configuration state within a VAR object).

Each IDENT object MAY be derived from one or more other base IDENT objects to form a directed graph. An IDENT which is not derived from any other is referred to as a "root" object. Any chain of derived IDENT objects SHALL NOT form a loop. The IDENT base graph provides a means to constrain its use in an object formal parameter (Section 3.4.1) and validate its use in an object reference given parameter (Section 3.1.2.1).

The definition of a IDENT consists of the following:

Name, Enumeration, Status, Reference, Description:
As defined in Common Object Fields.

Parameters:
A non-abstract IDENT definition MAY include formal parameters to be used when the IDENT is referenced. An abstract IDENT SHALL NOT include any formal parameters. Parameterized objects are discussed in Section 7. The formal parameters of an IDENT are fixed and SHALL NOT change between ADM revisions.

Abstract marking:
A boolean indication of whether this IDENT is considered abstract, meaning it cannot be used as an end value, or not. The abstract marking of an IDENT MAY change between ADM revisions, but only from abstract to non-abstract. This ensures that earlier state referencing the IDENT does not become invalidated.

Base references:
: An unordered list of object references to other IDENT objects from which this object is derived. The bases of an IDENT MAY change between ADM revisions, but only to add and not remove base references. This ensures that earlier state referencing the IDENT does not become invalidated.

As defined in this document, IDENTs can only be defined within an ADM.

The purpose of the abstract marking is to disallow specific objects in an IDENT hierarchy, typically those closest to the root object, from being referenced by values stored in CONST or VAR objects or produced by EDD objects. It would also be useful in a manager-side user interface to filter-out choices for IDENT objects which are marked as abstract. IDENT objects marked as abstract are expected to be used as the "base" for semantic types or other IDENT objects.

3.4.4. Externally Defined Data (EDD)

Externally defined data (EDD) objects represent data values that are produced based on a source external to the Agent itself. The Value Production occurs at the moment the value is needed, by either an Evaluation or a Reporting activity. The actual value could come from outside of the Agent proper, or be derived from data outside of the Agent.

The value production of an EDD SHOULD be nilpotent and have no side-effects in the processor. This property is not enforced by the Agent but requires consideration of the ADM designers, see Section 7.

The value produced by an EDD is allowed to, but not required to, change over time. Because EDDs can be referenced by condition expressions of Time-Based Rules (Section 3.4.9) or elsewhere, an Agent implementation could be optimized by allowing an EDD to indicate when its produced value _would_ change. It is an implementation matter for if and how an application can provide that indication.

For values managed entirely within the Agent use a Variable (VAR) or for constant-values use a Constant (CONST). For complex tabular data, use an EDD with a type which produces an ARI Table (TBL).

The definition of an EDD consists of the following:

Name, Enumeration, Status, Reference, Description:
: As defined in Common Object Fields.

Parameters:
An EDD definition MAY include formal parameters to be used when the EDD is used to produce a value. Parameterized objects are discussed in Section 7. The formal parameters of an EDD are fixed and SHALL NOT change between ADM revisions.

Type:
An EDD definition SHALL include the type of the value produced by the object, as described in Section 3.3. The type of an EDD is fixed and SHALL NOT change between ADM revisions.

As defined in this document, EDDs can only be defined within an ADM. Future capability could allow the use of EDDs within ODMs.

3.4.5. Constant (CONST)

A Constant (CONST) represents a named literal value, but unlike an Externally Defined Data (EDD) or Variable (VAR) a CONST always produces the same value. Examples include common mathematical values such as PI or well-known time epochs such as the UNIX Epoch. A CONST typed to produce a simple value can be used within an expression (see Section 6.7), where the object is used to produce a value at the moment of evaluation. A CONST can also be typed to produce an EXPR value to evaluate, MAC value to execute, or a RPTT value to generate reports.

The definition of a CONST consists of the following:

Name, Enumeration, Status, Reference, Description:
As defined in Common Object Fields.

Parameters:
A CONST definition MAY include formal parameters to be used during value production (Section 6.5.1). Parameterized objects are discussed in Section 7.13. Parameters for a CONST are only meaningful when the value uses actual parameters themselves containing one or more LABEL type, each referencing a formal parameter identifier.

Type:
A CONST definition SHALL include the type of the value produced by the object, as described in Section 3.3. The type of a CONST is fixed and SHALL NOT change for the lifetime of its containing namespace.

Value:
A CONST definition SHALL include the literal value produced during evaluation. The value of a CONST is fixed and SHALL NOT change for the lifetime of its containing namespace.

When a constant is not longer needed it SHALL be marked with a status of "deprecated" or "obsolete" rather than being removed (see Section 4.1.1 and Section 7.1). Allowing operators to define constants dynamically in an ODM means that a constant could be defined, removed, and then re-defined at a later time with a different type or value, which defeats the purpose of having constants so is prohibited above.

3.4.6. Control (CTRL)

A Control (CTRL) represents a predefined function that can be executed 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 outside of the Agent proper.

The execution of a CTRL SHOULD be idempotent and have no effect if executed multiple times in sequence. This property is not enforced by the Agent but requires consideration of the ADM designers, see Section 7.

Controls can be executed in a "one shot" manner as part of messaging from a Manager to an Agent. Network operators that wish to autonomously execute functions on an Agent may use a State-Based Rule (SBR) or Time-Based Rule (TBR). When an execution involves the ordered sequence of controls, a Macro (MAC) SHOULD be used instead of a more fragile use of CTRL directly.

The definition of a CTRL consists of the following:

Name, Enumeration, Status, Reference, Description:
As defined in Common Object Fields.

Parameters:
A CTRL definition MAY include formal parameters to be used when the CTRL is executed. Parameterized objects are discussed in Section 7. The formal parameters of a CTRL are fixed and SHALL NOT change between ADM revisions.

Result:
A CTRL definition MAY include the definition of a result. The result SHALL have a name and a type. The result MAY have a default value. The result of a CTRL is separate from the execution status as being successful or failed.

As defined in this document, CTRLs can only be defined within an ADM. Future capability could allow the use of CTRLs within ODMs if there was some mechanism to bind a CTRL definition to some platform-specific execution specification (_e.g._, a command line sequence).

3.4.7. Operator (OPER)

An Operator (OPER) represents a user-defined, typically mathematical, function that operates within the evaluation of an Expression (EXPR). It is expected that operators are implemented in the firmware of an Agent.

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 object type and Controls do not interact with operators.

The definition of an OPER consists of the following:

Name, Enumeration, Status, Reference, Description:
: As defined in Common Object Fields.

Parameters:
: An OPER definition MAY include formal parameters to be used when the OPER is evaluated. Parameterized objects are discussed in Section 7. The formal parameters of an OPER are distinct from the operands from the expression stack.

Operands:
: An OPER definition MAY include definitions of operand values to be popped from the expression stack when the OPER is evaluated. Each operand SHALL consist of a name, a type, and a cardinality. Any non-trivial OPER will have one or more operands. An OPER can have a non-fixed operand count which is based on a parameter value (_e.g._, an operator can average the top _N_ values from the stack, where _N_ is a parameter).

Result:
: An OPER definition SHALL include definition of a result value to be pushed onto the expression stack after the OPER is evaluated. The result SHALL have a name and a type. The result SHALL NOT have a default value.

As defined in this document, OPERs can only be defined within an ADM. Future capability could allow the use of OPERs within ODMs if there was some mechanism to bind an OPER definition to some platform-specific evaluation specification.

3.4.8. State-Based Rule (SBR)

A State-Based Rule (SBR) is a form of autonomy in which the Agent performs an action upon the change of state to meet a specific condition.

The execution model of the SBR is to evaluate the Condition (as often as necessary to handle changes in its expression evaluation) and when it evaluates to a truthy (Section 6.11.1) value and it has been no shorter than the Minimum Execution Interval since the last execution, the Action is executed. When the number of executions since the rule was enabled reaches the Maximum Execution Count the SBR is disabled. The execution occurs concurrently with any time processing and may take longer than the Minimum Execution Interval, so it is possible that multiple executions are requested to overlap in time.

Each SBR has an enabled state to allow rules to be retained in an ADM or ODM but not enabled during Manager-controlled time periods or under certain Manager-desired conditions. See Section 4.3 for details about what SBR-related controls are in the Agent ADM.

The definition of an SBR consists of the following:

Name, Enumeration, Status, Reference, Description:
: As defined in Common Object Fields.

Action:
: An SBR definition SHALL include an action in the form of a Macro (MAC). When triggered, the action execution SHALL be executed in accordance with Section 6.6 in an execution context with no parameters.

Condition:
: An SBR definition SHALL include a condition in the form of an Expression (EXPR). The condition SHALL be evaluated in accordance with Section 6.7 in an evaluation context with no parameters. The result of the condition SHALL be converted to a BOOL value after evaluation and used to determine when to execute the action of the SBR.

Minimum Execution Interval:
: An SBR definition SHALL include a minimum execution interval in the form of a non-negative TD value. The interval MAY be zero to indicate that there is no minimum. This value can be used to limit potentially high processing loads on an Agent.

An Agent MAY use this interval as a hint to also regulate the evaluation of the associated SBR condition. An Agent SHOULD use a threshold to ensure each SBR condition is evaluated at some maximum interval which can be less than this minimum execution interval. It is an implementation concern to determine when and how often to evaluate the condition, whether related to this interval or not. Users creating SBRs need to take this into consideration when configuring this parameter.

Maximum Execution Count:
An SBR definition SHALL include a maximum execution count in the form of a non-negative UVAST value. The count sentinel value zero SHALL be interpreted as having no maximum. This is _not_ a limit on the number of evaluations of the condition.

Initial Enabled:
An SBR definition MAY include an initial value for its enabled state. If not provided, the initial enabled state SHALL be true.

3.4.9. Time-Based Rule (TBR)

A Time-Based Rule (TBR) is a form of autonomy in which the Agent performs an action at even intervals of time.

The execution model of the TBR is to start a timer at the Start Time of the TBR ticking at an even Period; each time the timer expires the Action is executed. When the number of executions since the rule was enabled reaches the Maximum Execution Count the TBR is disabled. The execution occurs concurrently with any time processing and may take longer than the TBR Period, so it is possible that multiple executions are requested to overlap in time.

Each TBR has an enabled state to allow rules to be retained in an ADM or ODM but not enabled during Manager-controlled time periods or under certain Manager-desired conditions. See Section 4.3 for details about what TBR-related controls are in the Agent ADM.

The definition of a TBR consists of the following:

Name, Enumeration, Status, Reference, Description:
As defined in Common Object Fields.

Action:
A TBR definition SHALL include an action in the form of a Macro (MAC). When triggered, the action execution SHALL be executed in accordance with Section 6.6 in an execution context with no parameters.

Start Time:
A TBR definition SHALL include a start time in the form of a TIME (Section 4.2.5) value. A relative start time SHALL be interpreted relative to the absolute time at which the Agent is initialized (for ADM rules) or the rule is created (for ODM rules). The start time MAY be the relative time zero to indicate that the TBR is always active. This is _not_ a limit on the interval of evaluations of the condition.

Period:
A TBR definition SHALL include a period in the form of a positive TD value. The period SHALL NOT be zero but any non-zero small period is valid.

Maximum Execution Count:
A TBR definition SHALL include a maximum execution count in the form of a non-negative UVAST value. The count sentinel value zero SHALL be interpreted as having no maximum.

Initial Enabled:
A TBR definition MAY include an initial value for its enabled state. If not provided, the initial enabled state SHALL be true.

3.4.10. Variable (VAR)

A Variable (VAR) is a stateful store of a value in an Agent. The use of a VAR is similar to an EDD (Section 3.4.4) except that all the behavior of a VAR is entirely within an Agent, while the ultimate source of an EDD value is outside of the Agent.

The value production of a VAR into a value SHALL be nilpotent and have no side-effects in the processor.

The value produced by an VAR is allowed to, but not required to, change over time. Because VARs can be referenced by condition expressions of Time-Based Rules (Section 3.4.9) or elsewhere, an Agent implementation could be optimized by allowing a VAR to indicate when its produced value _would_ change. It is an implementation matter for if and how an application can provide that indication.

A VAR has an initializer, which is used at Agent initialization and to reset the VAR (see Section 4.3), but the VAR is otherwise stateful and will retain its last value between any actions which modify it.

The definition of a VAR consists of the following:

Name, Enumeration, Status, Reference, Description:
As defined in Common Object Fields.

Parameters:
A VAR definition MAY include formal parameters to be used during value production (Section 6.5.1). Parameterized objects are discussed in Section 7.13. Parameters for a VAR are only meaningful when the value uses actual parameters themselves containing one or more LABEL type, each referencing a formal parameter identifier.

Type:
An VAR definition SHALL include the data type of the value produced during evaluation, as described in Section 3.3. The type of a VAR is fixed and SHALL NOT change between ADM revisions.

Initializer:
An VAR definition MAY include an initializer in the form of an AMM value. The only times the initializer are needed are at Agent Initialization and when a CTRL is used to reset the state of the VAR. The initializer of a VAR MAY change between ADM revisions.

| NOTE: It is possible to specify an initializer value that does
| not match the type of the VAR. The VAR initializer will always
| be converted (Section 6.11) to the type of the VAR before
| assignment.

4. Application Data Models (ADMs)

An ADM is a logical entity for defining static AMM object instances, which are discussed in detail in Section 3.4. Each ADM exists as a separate namespace for its contained objects, but allows importing object _names_ from other ADMs to reuse them. Each Agent can support any number of ADMs at one time (subject to implementation limitations) and each Manager can operate with ADMs of different revisions to support diverse Agents.

The following subsections define what is present in an ADM generally and what objects necessary to operate a DTNMA Agent are present in two base ADMs.

4.1. ADM Definitions

An ADM is "static" in the sense that it is revision-controlled and a released revision of an ADM does not change. Besides AMM object definitions there are metadata and handling rules for the ADM itself, which are discussed in this section.

4.1.1. ADM Metadata

This section explains the purposes of the metadata fields of an ADM, while the specific syntax for how these fields fit into an ADM module is left to another document.

Module Name and Namespace:
: Both the module name and its namespace uniquely identify an ADM among all other possible ADMs, both well known and privately used. The module name takes the form of a file name (specific to the ADM encoding defined outside of this document), while the module namespace takes the form of an ARI (Section 3.1.4).

  These module identifiers are decomposed into two parts, described below: an organization ID part and a model ID part. A specific ADM encoding can, and likely will, have some amount of redundancy in how the module identifiers and organization-and-model identifiers are encoded. Because these identifiers are not allowed to change between revisions, any redundancy will at least not affect editing and updating of the ADM over time.

Organization Name and Enumeration:
: Each ADM definition SHALL reference an organization name. The organization name SHALL NOT change between ADM revisions. The organization name SHALL conform to the id-text ABNF rule of Section 3.4.1.

  Each ADM definition SHALL reference an organization enumeration. The organization enumeration SHALL NOT change between ADM revisions. The organization enumeration SHALL be a non-negative value in the domain of a signed 32-bit integer value.

Model Name and Enumeration:
: Each ADM definition SHALL contain a model name. That name SHALL be unique within the organization. The model name SHALL NOT change between ADM revisions. The model name SHALL conform to the id-text ABNF rule of Section 3.4.1.

  A model enumeration is an integer, associated with the whole ADM, which identifies the model within its organization just like its name. Model enumerations provide a stable and concise identifier for the binary encoded form of an ARI. Each ADM definition SHALL contain a model enumeration. The model enumeration SHALL NOT change between ADM revisions. The model enumeration SHALL be a non-negative value in the domain of a signed 32-bit integer value.

Revision History:
: Each ADM SHALL contain a history of dated revisions. At least one revision SHALL be present and mark the date at which the ADM was released for use. During development and testing an ADM need not have updated revisions, only when a release occurs should a revision be added.

Status:
: Each ADM definition SHOULD contain a status field. The valid status value of an ADM SHALL be one of "current", "deprecated", or "obsolete". These values are identical to the Status field of YANG Section 7.21.2 of [RFC7950]. In the absence of a status field, the status of the ADM SHALL be considered the same as the status of the ADM which contains it.

Reference:
: Each ADM definition SHOULD contain a reference field. A reference is a text string referring to a specification or other document which details the source or purpose of the ADM.

Description:
: Each ADM definition SHOULD contain a description field. A description is a text string explaining the purpose or usage of the ADM in a human-readable format.

Features:
: Each ADM definition MAY contain a set of feature definitions, see Section 4.1.2 for details. Each feature SHALL have a name that is unique within the namespace of the ADM. The name SHALL conform to the id-text ABNF rule of Section 3.4.1.

4.1.2. Features and Conformance

Following in the pattern of YANG features from Section 5.6.2 of [RFC7950] and SMIv2 conformance groups from [RFC2580], the AMM has the concept of ADM features and Agent conformance to those features. Each feature is a simple qualified name and each object in an ADM can be conditional on the conformance to a set of features.

In the same way that an Agent instance can choose to implement or omit any particular ADM (assuming its dependencies are satisfied), an Agent instance can choose to implement or omit particular features within an ADM. This allows more fine-grained control of what an Agent supports at runtime and also provides a standard mechanism for naming and indicating that support.

4.2. Contents of an AMM ADM

This base ADM is a necessary part of the AMM typing, execution, and evaluation models. Rather than having some Agent logic defined purely by specification, this document uses an "AMM" ADM to define semantic types and controls needed for normal Agent operations. The needed types are still set by specification and are unchanging within an ADM revision, but this avoids having a separate, intermediate typing system between the AMM-defined semantic types and the ARI-defined literal types. This is also in-line with how YANG [RFC6991] and SMIv2 [RFC2578] both rely on base modules for some core behavior.

4.2.1. Display Hint Root

Rather than using fixed enumerations for the display hint of a Named Type Use, the AMM uses an IDENT (Section 3.4.3) hierarchy, where each leaf object represents a specific form of display for one of the built-in types. The root IDENT object for this hierarchy is defined in this ADM, but the leaf objects will be managed outside the ADM. This follows the pattern described in Section 7.8 for extensible hints.

4.2.2. Type Introspection Objects

To allow reflecting the contents of semantic types within the AMM value system itself, in order to support type introspection of AMM objects, the AMM defines parameterized IDENT objects used to represent each of the semantic types (Section 3.3) available in the AMM. Each of the semantic type objects is derived from a base IDENT object, which is combined with the ARITYPE built-in type (Section 3.2) to produce a TYPEDEF that is able to capture all possible value types available to the AMM.

4.2.3. Simple Semantic Types

The most basic use of a semantic type is to provide additional meaning to simple types. None of these types associates a unit with the value, which it is expected that a derived type or an anonymous type (at the point of use) would add for additional clarity.

These are summarized below:

counter32 and counter64: An unsigned integer value with an arbitrary initial value which increments over time and wraps around the maximum value. These correspond with the same names defined in YANG [RFC6991] and SMIv2 [RFC2578].

gauge32 and gauge64: An integer value sampling some measurement

which can increase or decrease arbitrarily over time. These correspond with the same names defined in YANG [RFC6991] and SMIv2 [RFC2578].

timestamp: An absolute time at which an event happened. This corresponds with the same name defined in YANG [RFC6991] and SMIv2 [RFC2578].

4.2.4. Container Semantic Types

This section contains more complex semantic types which constrain the contents of a container (Section 3.2.2) so that the value as a whole has a specific semantic.

4.2.4.1. Expression (EXPR)

An Expression (EXPR) is an ordered collection of references to Operators or operands. An EXPR takes the form of a semantic typedef refining an AC to be a list of ARIs referencing OPERs, ARIs referencing evaluate-able objects (see Section 6.7), or literal value ARIs (with Simple Types). These operands and operators form a mathematical expression that is used to compute a resulting value.

The evaluation procedure of an EXPR is defined in Section 6.7. Expressions are used within an ADM for defining the initializer of a Variable (VAR) and for defining the condition of a State-Based Rule (SBR).

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 SHALL be represented in postfix order. Postfix notation requires no additional symbols to enforce precedence, always results in a more efficient encoding, and postfix engines can be implemented efficiently in embedded systems.

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

4.2.4.2. Macro (MAC)

A Macro (MAC) is an ordered collection of references to Controls or other Macros. A Macro takes the form of a semantic typedef refining an AC to be a list of ARIs referencing Controls or objects which produce other Macros.

The execution procedure of an MAC is defined in Section 6.6. Macros are used within an ADM for defining the action of a State-Based Rule (SBR) or Time-Based Rule (TBR).

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

4.2.4.3. Report Template (RPTT)

A Report Template (RPTT) is an ordered list of object references or expression values used as a source for generating items for report (Section 3.2.2.5) containers. A RPTT takes the form of a semantic typedef refining an AC to be a list of references to value-producing objects (VALUE-OBJ (Section 4.2.5)) or expressions (EXPR (Section 4.2.4.1)). An object which produces an RPTT can itself be parameterized so that the object flows down parameters as described in Section 3.1.2.1.

A RPTT can be viewed as a schema that defines how to generate and interpret a Report; they contain no direct values. RPTT values either defined in an ADM or configured between Managers and Agents in an ODM. Reports themselves are ephemeral and represented within ARI built-in type RPTSET, not as part of the AMM object model. The procedure for reporting on a RPTT is defined in Section 6.8.2.

RPTT values SHOULD be used within a CONST where possible. RPTT values MAY be used within a VAR where necessary. This makes correlating a RPT value with its associated RPTT easier over time. Rather than having a VAR object’s RPTT value changing over time, it is RECOMMENDED to deprecate earlier RPTT-producing CONST objects and create new objects.

4.2.4.4. Execution Target Type

A convenience typedef exec-tgt is defined to codify the type of values allowed to be used as input for an Execution (or within an Execution-Set (EXECSET) value) or produced by objects referenced as execution targets. The execution target type is defined to be either a direct CTRL reference, a direct MAC value, or a reference to a value-producing object which itself is typed as exec-tgt.

4.2.4.5. Evaluation Target Type

A convenience typedef eval-tgt is defined to codify the type of values allowed to be used as input for an Evaluation activity or produced by objects referenced as evaluation targets. The execution target type is defined to be either a direct SIMPLE value, a direct EXPR value, or a reference to a value-producing object which itself is typed as eval-tgt.

4.2.5. Type Unions

All of the literal types defined in Section 3.2 have a flat structure, with some types sharing the same value structure but using distinct built-in type code points to distinguish them (_e.g._, both BYTESTR and CBOR use the same byte-string value). In order to allow types to fit into a more logical taxonomy, the AMM defines some specific semantic typedefs to group literal types. These groups are not a strict logical hierarchy and are intended only to simplify the effort of a model designer when choosing type signatures.

These are summarized below:

TYPE-REF: The union of ARITYPE and TYPEDEF types for built-in types and named semantic types respectively.

INTEGER: The union of BYTE, UINT, INT, UVAST, and VAST types.

FLOAT: The union of REAL32 and REAL64 types.

NUMERIC: The union of INTEGER and FLOAT types.

PRIMITIVE: The union of NULL, BOOL, NUMERIC, TEXTSTR, and BYTESTR types. This matches any untyped literal value.

TIME: The union of TP and TD types.

SIMPLE: The union of PRIMITIVE, TIME, LABEL, CBOR, and ARITYPE types. This matches any non-container, non-reference-pattern literal value, both typed and untyped.

ANY: The union of LITERAL, OBJECT, and NAMESPACE value-class types (Section 3.2.4). This matches all values that can be in an ARI.

VALUE-OBJ: The union of CONST, EDD, and VAR reference types. This matches any reference to an object that can produce a value (Section 6.5).

NONCE: The union of BYTESTR, UINT64, and NULL types. This is used

by EXECSET and RPTSET values to correlate Agent-Manager messages (see Section 2.3).

ID-TEXT: A use of TEXTSTR type with semantics to align with the "name" field of Section 3.4.1 (_i.e._, the id-text ABNF rule).

ID-INT: A use of INT type with semantics to align with the "enumeration" field of Section 3.4.1 (_i.e._, no additional constraints needed).

4.3. Contents of an Agent ADM

While the AMM ADM described in Section 4.2 contains definitions of static aspects of the AMM, the DTNMA Agent ADM is needed to include necessary dynamic aspects of the operation of an Agent. This separation is also helpful in order to allow the dynamic behaviors of an Agent to be modified over time while the AMM definitions stay stable and unchanging.

4.3.1. Agent State Introspection

The Agent ADM contains the following EDD objects used to introspect the Agent’s state, all of which can change over time within an Agent.

* The ADMs supported by the Agent, including the unique name and revision of each. By indicating specific revision and supported feature set, the contained objects in each ADM can be derived. Because of this, the ADM-contained objects do not require additional introspection.

* The set of SBRs and TBRs in the Agent’s ODMs, along with controls to ensure a specific object is either present or absent. These are all conditioned on whether the Agent actually supports the built-in rule feature.

* The set of VARs in the Agent’s ODMs, along with controls to ensure a specific object is either present or absent.

* Visibility into the execution state(s) of an Agent, including counters for the total number of successful and failed executions.

* Counters for the total number of messages sent or received by the agent, including reception failures.

4.3.2. Macro Helper Controls

The Agent ADM contains a set of controls which implement behaviors to macro execution logic.

Branching Control: This control has a condition parameter to evaluate and two optional parameters to define sub-macros, one of which is executed depending upon the condition result truthy-ness.

Failure Catching Control: This control has one parameter of a macro to execute normally and a second parameter of a macro to execute on the condition that the normal execution fails for some reason.

Waiting Controls: This family of controls is to be embedded at the start (or anywhere within) a macro and pauses execution to wait on either: a specific absolute time, a relative time from start-of-control, or a condition to evaluate to truthy.

4.3.3. Basic Operators

The Agent ADM contains a set of operators which provide logical and mathematical functions to macro execution.

Numeric Operators: These perform operations related to negation, addition, subtraction, multiplication, division, and remainder (modulo) of their operands. These perform numeric promotion of their operands in accordance with Section 6.11.2.1.

Boolean Operators: These perform operations related to boolean NOT, AND, OR, and XOR of their operands. These perform boolean casting of their operands in accordance with Section 6.11.1.

Bitwise Operators: These perform bitwise NOT, AND, OR, and XOR operations on only unsigned integer operands.

Comparison Operators: These perform pairwise comparison between their operands. Equality and inequality can be performed on any operand types, but ordered comparison (_e.g._, greater than) can only be performed on numeric operands. Equality comparison distinguishes between typed and untyped literals in their operands in accordance with Section 6.12.

Table Filtering: This operator is used to process tables produced within an expression to filter by row contents and specific columns. This is an example of a parameterized operator because the parameters control the filtering while the operand is the table-to-be-filtered.

Container Extraction: These operators are used to extract individual values from within an AC, AM, or TBL container. The specific item to be extracted is identified by operator parameter(s) such as AC index, AM key, or TBL row/column indices.

5. Operational Data Models (ODMs)

An ODM is a logical entity for containing AMM objects, similar to an ADM (Section 4) but in an ODM the objects are not static. An ODM's objects can be added, removed, and (with some restrictions) modified during the runtime of an Agent. Like an ADM, each ODM exists as a separate namespace for its contained objects and an Agent can contain any number of ODMs.

Some object types, those which require implementation outside of the Agent proper, are not available to be created in an ODM. These include the CTRL, EDD, and OPER.

The actions for inspecting and manipulating the contents of an ODM are available through EDDs and CTRLs of the Agent ADM (Section 4.3.1).

6. Processing Activities

This section discusses logic and requirements for processing of AMM objects and values. Each subsection is a separate class of processing that is performed by an Agent.

A Manager (or any other entity) MAY perform some of the same processing, _e.g._ evaluating an expression, in order to validate values or configurations before sending them to an Agent. That kind of behavior is effectively creating a "digital twin" of the managed Agent to ensure that the processing will behave as expected before it is sent. For this reason, the subject noun used in all of these activities is the "processor".

6.1. Agent Initialization

The initialization of the Agent state can be associated with a power-on event or, due to the use of volatile memory, can be an explicit activity initiated from outside the Agent runtime. If volatile memory is used the contents of the ODMs on an Agent will be present for the initialization procedure; otherwise the ODMs will be considered empty or absent.

The procedure to initialize an Agent is as follows:

1. All ADM-defined VAR objects SHALL have their value set to one of the following:

    * If an Initializer is defined for the VAR, the value is the result of evaluating the associated Initializer expression and then converting (Section 6.11) to the VAR type.

* Otherwise, the value is undefined.

Any ODM-defined VAR objects MAY retain their state.

2. All ADM-defined TBR and SBR objects SHALL have their Enabled state set to the Initial Enabled value. Any ODM-defined TBR and SBR objects MAY retain their Enabled state. Any rules which are enabled are ready for processing.

6.2. ARI Resolving

Within an ADM, ARIs present in the various fields of object definitions are URI References, which can take the form of relative references (see Section 4.2 of [RFC3986]). Any ARIs within an ADM definition SHALL be handled as URI References and resolved in accordance with the procedure of Section 5 of [RFC3986] with the namespace reference of the containing ADM used as a base URI.

One side effect of using the namespace reference as a base is that ARIs within an ADM _do not_ require a URI scheme part.

6.3. Object Dereferencing

An Object Reference Values contains an object path and a parameter part. Dereferencing an OBJECT value uses the object path to look up a specific defined object available to the agent.

The process of dereferencing a value is as follows:

1. The value has to contain an object reference (_i.e._, it matches the built-in type OBJECT). If the value is not an object reference, this procedure stops and is considered failed.

2. The OBJECT value namespace (organization ID, model ID, and model revision) is used to search for a defined ADM or ODM namespace. A text form organization ID or model ID SHALL be compared within the UTF-8 character set in accordance with [RFC3629]. An integer form organization ID or model ID SHALL be compared numerically. If present, a model revision SHALL be compared to the latest revision of an ADM as either UTF-8 text or an internal decoded date form with equivalent comparison behavior. If no corresponding namespace is available, this procedure stops and is considered failed.

3. Within the namespace the object type and object name (whether text or enumeration) is used to search for a specific defined object. A text form object name SHALL be compared within the UTF-8 character set in accordance with [RFC3629]. An integer

object name namespace SHALL be compared numerically. If no corresponding object is available, this procedure stops and is considered failed.

6.3.1. Parameter Handling

An Object Reference Values contains an object path and a parameter part. The parameter part of an OBJECT value represents the given parameters (Section 3.1.2.1) being used. The given parameters are present either as a (possibly empty) ARI list or an ARI map. Due to nuances of the AMM value system, the given parameters are not themselves either AC or AM values but similar to untyped ARI values.

The process to validate and normalize _given parameters_ against an object's _formal parameters_ to produce _actual parameters_ is as follows.

1. For each formal parameter, the processor performs the following:

   If the given parameters are a list, the formal parameter is correlated to the list by its position in the formal parameters list. If the given parameters list does not contain a corresponding position the given parameter is treated as the undefined value. If the last formal parameter is a Sequence (Section 3.3.7), it can correlate with multiple given parameters.

   If the given parameters are a map, the formal parameter is correlated to a map key by either its position (as an integer) or its name (as a text string) but not both. If both integer and name are present in the given parameters map the procedure stops and is considered failed. If the given parameters map does not contain a corresponding key the given parameter is treated as the undefined value.

2. If any of the given parameters is not correlated with a formal parameter the procedure stops and is considered failed.

3. For each correlated pair of formal parameter and given parameter(s), the processor performs the following:

   a. If the given parameter is undefined (whether explicitly or implicitly) and the formal parameter defines a default value, that default is used as the actual parameter value. If there is no default value, the actual parameter is left as the undefined value.

b. If the given parameter is a TYPEDEF and the object reference itself has a parameter, the given parameter is treated as the result of a type conversion (Section 6.11.3) to the semantic type of the TYPEDEF. If the conversion fails this procedure stops and is considered failed.

c. The given parameter is converted to an actual parameter using the type of the formal parameter in accordance with Section 6.11. If the conversion fails, this procedure stops and is considered failed.

| The TYPEDEF conversion behavior in step 3.b acts as an explicit
| type cast within a given parameter which allows explicit tie-
| breaking for type unions in the corresponding formal parameter.

The actual parameters resulting from this procedure are intended to be able to be looked up by an implementation either by ordinal position in the formal parameters list or by unique name of the formal parameter. It is an implementation matter whether or not to provide both accessing methods and the specifics of how, for example, and EDD or CTRL runtime accesses actual parameter values.

An implementation MAY perform deferred "lazy" processing of any of the above steps, causing a failure when the actual parameter value is needed. One caveat about deferred processing is that it will not fail if the parameter is unused, which is not necessarily a problem but could mask other issues in whatever provided the given parameters.

6.4. Object Reference Pattern Matching

An Object Reference Pattern SHALL be considered to match an object (instance) on an Agent when each component of the object identity matches each level of the object’s containment structure according to all of the following:

1. The organization ID pattern matches either the text name or integer enumeration of the organization containing the object.

2. The model ID pattern matches either the text name or integer enumeration of the model containing the object.

3. The object type pattern matches either the text name or integer enumeration for the type of the object.

4. The object ID pattern matches either the text name or integer enumeration of the object itself.

Each corresponding part of the pattern and corresponding object identity match according to the following:

Single Value: A pattern part of a single value SHALL match only the exact same value from the object identity part. A text value matches only the text name of an object and an integer value matches only the integer enumeration of an object.

Range: A pattern part of a range SHALL match object identity part within one of its integer intervals. The range is not able to match objects by text name.

Wildcard: A pattern part of a wildcard SHALL match any object identity part.

This procedure allows matching existing and even not-yet-defined future objects, for example when checking permission while defining new objects in an ODM.

This specification does not define a procedure for matching an Object Reference Pattern with an object reference value.

6.5. Value Production

Value production can be thought of as a common behavior used for Execution, Evaluation, and Reporting activities. Within the AMM the following entities have a value production procedure: CONST, EDD, and VAR object references.

This activity relies on an object reference value to have been dereferenced in accordance with Section 6.3 and its parameters handled in accordance with Section 6.3.1. After that, each of the object types is treated differently as defined in the following subsections.

6.5.1. CONST and VAR Objects

Both CONST and VAR objects act as a store of a single literal value within the Agent. Formal parameters on either CONST or VAR objects are applicable only when the objects store a value which itself contains parameters with at least one LABEL type.

The value production for these objects takes the stored value from the object and augments it by label substitution based on the following:

1. The processor checks whether the ACL contains a "produce" permission for the current context and object. If no such permission is present this procedure stops and is considered failed.

2. The processor identifies all LABEL type within the stored value, descending into container (Section 3.2.2) contents and object reference parameter contents as necessary.

3. If any LABEL text is not present in the formal parameters of the value-producing object then this procedure stops and is considered failed.

4. For each LABEL value the corresponding actual parameter is not the undefined value, the LABEL value is replaced by the actual parameter.

This augmentation has no effect on the stored value, it occurs only in the produced value. It is valid both for an actual parameter to have no substitution occur with its value and for an undefined actual value not be sued in substitution.

6.5.2. EDD Objects

For EDD objects, the actual parameters are used by the underlying implementation to produce the value in an arbitrary way. The produced value is typically either a SIMPLE (Section 4.2.5) value or an ARI Table (Section 3.2.2.3).

The value production for these objects occurs outside of the Agent proper within an implementation of the EDD being produced from as follows:

1. The processor checks whether the ACL contains a "produce" permission for the current context and object. If no such permission is present this procedure stops and is considered failed.

2. The initial state of the Result Storage is the undefined value to handle cases of application failure.

3. The request for production is forwarded to the application which implements the EDD, with additional context listed below. It is an implementation matter and author consideration (Section 7) to enforce that the produced value is consistent with the type of the object.

The context given to the EDD implementation is the following:

ACL Information:
The ACL groups and permissions for this production.

Object Path:
This gives visibility into the EDD object reference which was dereferenced during the production.

Actual Parameters:
The set of actual parameters used for the production.

Result Type:
An indication of the produced type for the object.

Result Storage:
The result of the production is placed here before completion.

6.6. Execution

Within the AMM only two entities can be the target of an execution procedure: controls and macros. Controls are executed by reference, while macros are executed both by value and by reference. This means the execution target value SHALL match the exec-tgt (Section 4.2.4.4) semantic type.

The procedure for executing is divided into phases to ensure that it does not fail due to invalid references or produced values after some controls have already been executed. The phases are processed as follows:

1. In the expansion phase the target value is processed to dereference all references, handle all parameters, and expand any produced values.

   If the target is a literal value, the following is performed:

   a. The value needs to match the MAC (Section 4.2.4.2) semantic type. If it does not match, this procedure stops and is considered failed.

   b. The processor then iterates through all elements of the MAC value and performs the expansion step on each in turn. If any sub-expansion fails, this procedure stops and is considered failed.

   If the target is an object reference, the following is performed:

a. The value is dereferenced in accordance with Section 6.3 and its parameters are handled in accordance with Section 6.3.1. If either fails, this procedure stops and is considered failed.

b. If the target object is a value-producing object, a value is produced in accordance with Section 6.5. This includes substitution of any LABEL parameters within the value.

c. The processor then performs the expansion step on the produced value. If sub-expansion fails, this procedure stops and is considered failed.

After successful expansion the target is either a dereferenced CTRL object, or a (possibly nested) macro expanded to contain only dereferenced CTRL objects. If expansion has failed, reporting still occurs in accordance with Section 6.6.3 using the execution target which failed to expand as the report source.

2. The processor then executes the top-level expanded value as either a macro in accordance with Section 6.6.1 or as a control in accordance with Section 6.6.2.

6.6.1. Expanded MAC Values

The execution of an Macro (MAC) value after expansion is as follows:

1. The processor iterates through all items of the expanded MAC in order and performs the following:

   If the item is an CTRL-REF it is executed in accordance with Section 6.6.2. If the execution fails, this procedure stops and is considered failed.

   Otherwise the item is an expanded sub-macro and it is executed in accordance with this procedure.

An effect of this procedure is that if any referenced CTRL fails during execution the processing fails immediately and subsequent CTRLs or MACs are not executed.

6.6.2. CTRL Objects

This activity relies on an object reference value to have been dereferenced in accordance with Section 6.3 and its parameters handled in accordance with Section 6.3.1.

The execution of a Control (CTRL) object occurs outside of the Agent proper within an application implementation of the CTRL as follows:

1. The processor checks whether the ACL contains an "execute" permission for the current context and object. If no such permission is present this procedure stops and is considered failed.

2. The initial state of the Result Storage is the undefined value to handle cases of application failure. This enables an execution to fail by simply leaving the initial result state in place.

3. The request for execution is forwarded to the application which implements the CTRL, with additional context listed below. To indicate that an execution has succeeded, the application SHALL set the Result Storage value. It is an implementation matter and author consideration (Section 7) to enforce that the result value is consistent with the result type.

The context given to the CTRL implementation is the following:

Manager:
: The manager which directly caused this execution, if available, is provided as context.

ACL Information:
: The ACL groups and permissions for this execution.

Object Path:
: This gives visibility into the CTRL object reference which was dereferenced during the execution.

Actual Parameters:
: The set of actual parameters augmented for the execution.

Result Type:
: An indication of the result type for the object.

Result Storage:
: The result of the execution is placed here before completion.

| The Agent ADM (Section 4.3) includes a wrapper CTRL "catch" which is used to ignore possible failures of specific executions and allow MAC processing to continue.

6.6.3. Execution Reporting

When an execution context is associated with a Manager endpoint and has a Correlator Nonce which is not null, after the execution of an individual CTRL has finished (whether successful or not) or an execution expansion fails the processor SHALL generate a RPTSET to be sent to that associated Manager containing:

Correlator Nonce: Copied from the execution context

Reference Time: The current timestamp

Report List: A single report containing:

Source: Copied from the CTRL reference which was executed

Generation Time: The zero-duration

Items: A single item copied from the result value, which is undefined if execution failed

The generated RPTSET can then be aggregated by the processor in accordance with Section 6.9.3. Whether or not any aggregation occurs is an implementation matter.

6.7. Evaluation

Within the AMM the following entities can be the target of an evaluation procedure: references to value-producing objects, OPERs, and TYPEDEFs and EXPR or SIMPLE literal values.

> | For the purposes of these procedures, it is important to distinguish between an EXPR _value_ and a reference to a value-producing object which is typed to produce an EXPR value.

The procedure for evaluation is divided into phases to ensure that it does not fail due to invalid references or produced values after some expressions have already been evaluated. The phases are processed as follows:

1. In the expansion phase the target value is processed to dereference all references, handle all parameters, and expand any produced values.

   If the target is a literal value, the following is performed:

a. The value needs to match the SIMPLE (Section 4.2.5) or EXPR (Section 4.2.4.2) semantic type. If it does not match, this procedure stops and is considered failed.

b. If the value is an EXPR, the processor then iterates through all elements of the EXPR value and performs the expansion step on each in turn. If any sub-expansion fails, this procedure stops and is considered failed.

If the target is an object reference, the following is performed:

a. The value is dereferenced in accordance with Section 6.3 and its parameters are handled in accordance with Section 6.3.1. If either fails, this procedure stops and is considered failed.

b. If the target object is a value-producing object, a value is produced in accordance with Section 6.5. This includes substitution of any LABEL parameters within the value.

c. The processor then performs the expansion step on the produced value. If sub-expansion fails, this procedure stops and is considered failed.

After expansion the target is either a SIMPLE value, or a (possibly nested) expression expanded to contain only SIMPLE or ARITYPE values, or dereferenced OPER or TYPEDEF objects.

2. If the expanded evaluation target is already a SIMPLE value, then that is the result of the evaluation. Otherwise, the expanded expression is evaluated in accordance with Section 6.7.1.

6.7.1. Expanded EXPR Values

The reduction of an Expression (EXPR) value after expansion is as follows:

1. Any sub-expressions are first reduced to their result values which are substituted back into the corresponding expression item. If any sub-evaluation fails this procedure stops and is considered failed. At this point the expression consists of only SIMPLE or ARITYPE values, the result value of sub-expression reduction, or dereferenced OPER or TYPEDEF objects.

2. An empty value stack is initialized for this reduction.

3. The expression is treated as a Reverse Polish Notation (RPN) sequence, where the following is performed on each item in the AC in sequence:

   If the item is an ARITYPE value or dereferenced OPER or TYPEDEF object it is evaluated in accordance with Section 6.7.4, Section 6.7.2 or Section 6.7.3 respectively. If the evaluation fails, this procedure stops and is considered failed.

   Otherwise, the item is pushed onto the stack.

4. After RPN processing if the value stack is empty or has more than one item, this procedure stops and is considered failed. Otherwise, the result of the evaluation is the single literal value in the stack.

One effect of this procedure is that if any referenced values cannot be produced the procedure fails before any OPER is evaluated. Another effect of this procedure is that if any referenced OPER fails during evaluation or any value production fails the EXPR processing fails immediately and subsequent OPER values, EXPR values, or VALUE-OBJ references are not evaluated.

6.7.2. OPER Objects

This procedure applies only during the evaluation of a containing expression (Section 6.7.1); an OPER cannot be evaluated in isolation.

The evaluation of an OBJECT value referencing a Operator (OPER) is as follows:

1. The value is dereferenced in accordance with Section 6.3 and its parameters are handled in accordance with Section 6.3.1. If either fails, this procedure stops and is considered failed.

2. The processor passes the evaluation on to the underlying implementation of the OPER being evaluated.

   The context available to the OPER implementation is the following:

   Object Path:
      This gives visibility into the OPER object reference which was dereferenced during the evaluation.

   Parameters:
      The set of actual parameters augmented for the evaluation.

Expression Stack:
The operands are popped from this stack and the result is pushed here before completion.

If the evaluation procedure fails, the failure SHALL propagate up to any expression evaluation.

6.7.3. TYPEDEF Objects

This procedure applies only during the evaluation of a containing expanded expression; a TYPEDEF object cannot be evaluated in isolation. The evaluation of a TYPEDEF is handled similarly to a unary OPER but it occurs entirely within the Agent and does not rely on an object-specific implementation.

The evaluation of a TYPEDEF object is as follows:

1. If the TYPEDEF value itself has no parameters, the input value is popped from the stack. If the TYPEDEF value itself has one parameter, the input value is that parameter. If the TYPEDEF value itself has more than parameter, this procedure stops and is considered failed.

2. The result value is a type conversion (Section 6.11.3) on the input value. If the conversion fails this procedure stops and is considered failed.

3. The result value is pushed onto the stack.

6.7.4. ARITYPE Values

This procedure applies only during the evaluation of a containing expanded expression; an ARITYPE value cannot be evaluated in isolation. The evaluation of an ARITYPE is handled similarly to a TYPEDEF but with no possibility of a parameterized conversion.

The evaluation of an ARITYPE value is as follows:

1. The input value is popped from the stack.

2. The result value is a type conversion (Section 6.11) on the input value. If the conversion fails this procedure stops and is considered failed.

3. The result value is pushed onto the stack.

6.8. Reporting

This reporting activity operates on a single source value, which can be either a literal or an object reference value, and a single destination Manager identity.

Reporting can be caused by another activity internal to an Agent, including the implementation of its Agent ADM (Section 4.3), or can be requested by managed applications interfacing with the Agent. The Agent ADM can include control(s) that cause this reporting activity to be performed based on more complex parameters (_e.g._ identifying multiple destination managers), which would result in multiple independent reporting activities occurring within the Agent. This allows each reporting activity and its associated value productions to be access controlled for a specific destination Manager.

The reporting procedure is composed of the following phases:

1. Obtain a Report Template (RPTT) value from the source based on the following:

   If the source matches the RPTT type that source is used directly as the template.

   Otherwise, if the source matches the VALUE-OBJ a value is produced from the object in accordance with Section 6.8.1. If that production fails, this procedure stops and is considered failed.

   Otherwise, the source is invalid so this procedure stops and is considered failed.

2. Mark a timestamp to be used for generating the RPTSET.

3. Transform items of that RPTT into items of a report in accordance with Section 6.8.2. This transformation cannot fail, but can result in undefined report items.

4. Use those report items to generate a full RPTSET value to be sent to the destination Manager accordance with Section 6.8.3

If failures occur in this procedure no RPTSET is generated but a parent execution which caused the reporting will itself fail and potentially provide feedback to a Manager.

Each RPTSET generated by this activity, as opposed to control execution (Section 6.6.2), has nonce of null because they can be destined for an arbitrary Manager rather than whatever Manager caused

an associated execution context. This means that there is nothing visible to the Manager which can directly correlate to the execution which caused the reports to be generated.

6.8.1. Value-Producing Source Reference

Obtaining a report template from a reference to a value-producing object is as follows:

1. The source is dereferenced in accordance with Section 6.3 and its parameters are handled in accordance with Section 6.3.1. If either fails, this procedure stops and is considered failed.

2. The value is produced from the object in accordance with Section 6.5. This step includes substitution of any LABEL parameters within the value.

3. If that produced value does not match the RPTT type, this procedure stops and is considered failed.

This procedure is not recursive, so any object references (or any other top-level values) of the produced RPTT are left unchanged by this procedure.

6.8.2. Transforming a Report Template

Transforming a RPTT value, which is structured as an AC, into a report item list is as follows:

1. A report item list is initialized for this template containing the same number of items as the RPTT itself. All initial items of the report are the undefined value.

2. The processor iterates through all items of the template, performing the following:

   If the template item matches the EXPR type the associated report item is replaced by the result of evaluation in accordance with Section 6.7. If the evaluation fails the undefined value is left as the report item.

   Otherwise, if the template item matches the VALUE-OBJ type the associated report item is replaced by the value produced in accordance with Section 6.5. If the production fails the undefined value is left as the report item.

   Otherwise, the template item cannot be reported on and the undefined value is left as the report item.

One effect of this procedure is that if any item of the RPTT cannot be reported on, the undefined value is used as a sentinel and the other report items are still generated.

6.8.3. Generating a RPTSET

Once report items are assembled, the Agent generates a RPTSET for the destination Manager containing:

Correlator Nonce: The null value

Reference Time: The timestamp marked before transforming the report template

Report List: A list with a single report containing:

Source: Copied from the source of this reporting activity

Generation Time: The zero-duration time difference

Items: A list of items, based on the source, as defined in one of the following subsections

The generated RPTSET can then be aggregated by the processor in accordance with Section 6.9.3. Whether or not any aggregation occurs is an implementation matter outside of this activity.

6.9. Agent-Manager Message Handling

6.9.1. Execution-Set Aggregation

Managers SHOULD aggregate multiple Execution-Set (EXECSET) values associated with the same Agent and Correlator Nonce into a single Execution-Set. The aggregation MAY be based on a size limit (_e.g._, number of targets), time limit, or an event (_e.g._, network availability). This avoids the overhead of transport and processing multiple executions on the same Agent, and due to the requirements in Section 6.9.2 makes no difference to (lack of) guarantees in execution order.

6.9.2. Execution-Set Processing

An Agent SHALL process an Execution-Set through the independent Execution of each item in the target list. Execution order is not guaranteed and failures on one target do not affect other target, so targets MAY be executed in any order or concurrently. This is not the same behavior as the execution of a macro, where execution of items is ordered and a failure of any execution causes subsequent

items to not be executed.

6.9.3. Reporting-Set Aggregation

Agents SHOULD aggregate multiple Reporting-Set (RPTSET) values associated with the same Manager and Correlator Nonce into a single Reporting-Set. The aggregation MAY be based on a size limit (_e.g._, number of reports or number of total report items), time limit, or an event (_e.g._, network availability or power-saving wake-up). This avoids the overhead of transport and processing multiple messages on a Manager and improves timestamp compression in the reports, but it does require that all of the items are associated with the same manager and nonce.

6.9.4. Reporting-Set Processing

A Manager SHALL process each report within a Reporting-Set independently. Failures in processing any one report do not affect other reports, so reports MAY be processed in any order or concurrently. After using a Report Template to correlate report items with source objects, a Manager SHALL treat each (timestamp, object, item value) tuple independently from its containing Reporting-Set or Report.

6.10. Type Matching

Type matching is done through pattern matching and does not affect the AMM value. AMM values are not strictly typed, and as long as an AMM value matches the pattern for a type, that value can be used where that type is needed. If there is any overlap in the patterns for different semantic types, then there will be ambiguity in the sense that the same value can be used as different types.

6.10.1. Built-In Types

The matching of a built-in literal type to any object reference value SHALL be considered to fail. The built-in LITERAL type SHALL match any typed or untyped literal value.

The matching of a built-in literal type to a typed literal value as follows:

1. If the value type differs from the built-in type, the match fails.

2. Otherwise, the literal type is matched to the value primitive as defined below.

The matching of a built-in literal type to an untyped literal value as follows:

NULL: This type only matches the null primitive value.

BOOL: This type only matches the true and false primitive values.

BYTE: This type only matches uint primitive values in the domain 0 to 2^8-1 inclusive.

INT: This type only matches int primitive values in the domain -2^31 to 2^31-1 inclusive.

UINT: This type only matches uint primitive values in the domain 0 to 2^32-1 inclusive.

VAST: This type only matches int primitive values in the domain -2^63 to 2^63-1 inclusive.

UVAST: This type only matches uint primitive values in the domain 0 to 2^64-1 inclusive.

REAL32: This type only matches float primitive values in the domain of a 32-bit [IEEE.754-2019] floating point number.

REAL64: This type only matches float primitive values in the domain of a 64-bit [IEEE.754-2019] floating point number.

TEXTSTR: This type matches tstr primitive values.

BYTESTR: This type matches bstr primitive values.

TP: This type matches int primitive values, and float values truncated to fixed precision.

TD: This type matches int primitive values, and float values truncated to fixed precision.

LABEL: This type matches int and tstr values, where the text also matches the id-text ABNF rule of Section 3.4.1. Implementations MAY relax or cache the matching condition for LABEL text values to avoid extra processing.

CBOR: This type matches bstr primitive values that are non-empty and contain a well-formed CBOR item. Implementations MAY relax or cache the well-formed-ness condition for CBOR values to avoid extra processing.

ARITYPE: This type matches int and tstr values.

Relaxing some built-in type conditions is done at the risk that users will supply invalid values that will not be caught until significant other processing has already occurred. If an Agent implementation relaxes these conditions it is RECOMMENDED to use only Managers which enforce the conditions in order to block invalid values before they arrive at an Agent.

The matching of a built-in object reference type to any literal value SHALL be considered to fail. The built-in OBJECT type SHALL match any object reference values.

The matching of a built-in object reference type to an object reference value SHALL be considered to succeed if the object reference value Type ID is identical to the type.

6.10.2. Semantic Types

The matching of an input value to each class of semantic type (Section 3.3) is as follows:

Named Type Use: Matching for a named type use SHALL be identical to the matching for the type being named, whether that is TYPEDEF or built-in.

Uniform List: Matching for this class SHALL require the value to be an AC, with an item count optionally constrained by minimum and maximum size from the type, and with each item of the AC itself matching the specific sub-type for the list.

Diverse List: Matching for this class SHALL require the value to be an AC, with an item count optionally constrained by minimum and maximum size from the type, and with each item of the AC itself matching the specific sub-type or sequence for the list.

Uniform Map: Matching for this class SHALL require the value to be an AM, with a size optionally constrained by minimum and maximum size from the type, and with each key and value of the AM itself matching the specific respective sub-type for the map.

Table Template: Matching for this class SHALL require the value to

be an TBL, with a column count matching exactly the number of columns present in the table template and each row containing items matching the column-specific sub-type for the template. If the table template contains a limit on minimum or maximum size, the row count SHALL conform with those limits to match. If the table template contains a key column or unique column-set then all rows SHALL satisfy the uniqueness of those constraints to match.

Type Union: The matching for a type union SHALL be performed as follows:

1. The underlying literal types usable with a semantic type are obtained by recursively flattening the type union(s) down to built-in types and removing duplicate built-in types after the first instance of them. This defines a list of acceptable built-in types for the result.

2. The input value is matched to each built-in type in the list, and the first successful match results in a success of matching the whole semantic type. If none of the built-in types can successfully match the input value, the match is considered failed.

3. If successful, the specific base type which matched is also part of the result of this procedure.

6.11. Type Conversion

The type system of the AMM allows conversions of values between different literal and semantic types in a way which is supposed to preserve the "meaning" of the value.

In some cases, type conversion is performed implicitly by the Agent while other cases the conversion is explicitly part of an expression. One example of implicit casting is during Parameter Handling to ensure each processed parameter meets the formal parameter type signature. Another example of implicit conversion is for numeric operators in the Agent ADM (Section 4.3.3).

A special case which applies to all conversions is that the undefined value is passed-through as the result value unconditionally.

6.11.1. BOOL Type

The AMM has the concepts of "truthy" and "falsey" as being the result of casting to BOOL type. Similar to the ToBoolean() function from [ECMA-262], the AMM casting treats the following as falsey and every other value except undefined as truthy.

* The null value (of NULL)
* The false value (of BOOL)
* Zero value of BYTE, UINT, INT, UVAST, and VAST
* Positive and negative zero, and NaN values of REAL32 and REAL64
* Empty value of TEXTSTR and BYTESTR (note that CBOR cannot be empty and is excluded here)
* Zero value of TD (note that TP is excluded here)

When casting a value to BOOL type, the processor SHALL use the result value false if the original value is falsey and true if the original value is not undefined. The undefined value is passed-through for a boolean conversion (just like other types).

6.11.2. NUMERIC Types

The casting of a value to a NUMERIC type is intended to easily allow mixed-type expressions while keeping the number of operators and parameter unions small.

When casting a value to an INTEGER type from any other NUMERIC type, the processor SHALL perform the following:

1. If the input is one of the FLOAT types and is not finite, the conversion is considered failed.
2. If the input is one of the FLOAT types, the value is truncated to an integer by rounding toward zero.
3. If the input value is outside the domain of the output type, the conversion is considered failed.

When casting a value to an FLOAT type from any other NUMERIC type, the processor SHALL perform the following:

1. If the input value is outside the domain of the output type, the conversion is considered failed.

6.11.2.1. Numeric Promotion

While the earlier discussion of numeric type casting is about converting from an input type to an output type, the concept of a type promotion is about finding a "least compatible type" which can accommodate most, if not all, of the input type range. Converting to a promoted type is called an "up" conversion, and from a promoted type a "down" conversion.

The promotion order for NUMERIC types is as follows:

* A promoted type has a larger span of values (the difference between largest and smallest representable value).

* A promoted type can gain signed-ness but not lose it.

* A promoted type can lose precision for some values.

This promotion logic does not guarantee that an up-conversion will always succeed (_e.g._ some large UVAST values will not fit within a VAST or REAL32) but does provide a strict ordering for finding a compatible type between two NUMERIC values. The "least compatible type" between two types SHALL be defined as the smallest up-conversion that will accommodate the input types, as indicated in Table 5. This is almost a strict ordering except for the conversion of INT and UVAST to VAST to accommodate both the signed-ness and the size of the inputs.

	BYTE	UINT	INT	UVAST	VAST	REAL32	REAL64
BYTE	BYTE	UINT	INT	UVAST	VAST	REAL32	REAL64
UINT		UINT	INT	UVAST	VAST	REAL32	REAL64
INT			INT	VAST	VAST	REAL32	REAL64
UVAST				UVAST	VAST	REAL32	REAL64
VAST					VAST	REAL32	REAL64
REAL32						REAL32	REAL64
REAL64							REAL64

Table 5: NUMERIC Type Promotion

6.11.3. Semantic Types

The converting of an input value to each class of semantic type (Section 3.3) is as follows. Similar to built-in type conversion, each semantic type class has the potential to change a value as needed to conform to type limitations.

Named Type Use:

Uniform List: Converting to this class SHALL require the value to be an AC; if the input is not an AC this procedure stops and is considered failed. If the uniform list contains limits on the number of items and the value does not satisfy those limits this procedure stops and is considered failed. For each item of the input value the uniform sub-type SHALL be used to convert to a corresponding output value; if any of those conversions fail this procedure stops and is considered failed.

Diverse List: Converting to this class SHALL require the value to be an AC; if the input is not an AC this procedure stops and is considered failed. If the diverse list contains limits on the number of items and the value does not satisfy those limits this procedure stops and is considered failed. For each item of the input value the corresponding diverse list sub-type SHALL be used to convert to a corresponding output value; if any of those conversions fail this procedure stops and is considered failed.

Uniform Map: Converting to this class SHALL require the value to be an AM, if the input is not an AC this procedure stops and is considered failed. If the uniform map contains limits on the number of pairs and the value does not satisfy those limits this procedure stops and is considered failed. For each key--value pair of the input value the uniform sub-types SHALL be used to convert to a corresponding output pair; if any of those conversions fail this procedure stops and is considered failed. If the conversion results in any duplicate keys this procedure stops and is considered failed.

Table Template: Converting to this class SHALL require the value to be a TBL, if the input is not an TBL this procedure stops and is considered failed. If the table template contains limits on the number of rows and the value does not satisfy those limits this procedure stops and is considered failed. If the value contains a different number of columns from the table template this procedure stops and is considered failed. For each item across each row of the input the corresponding column sub-type SHALL be used to convert to a corresponding output table item; if any of those conversions fail this procedure stops and is considered failed.

Type Union: Converting a value for a type union SHALL be performed as follows:

1. The underlying built-in types usable with a semantic type are obtained by recursively flattening type union(s) down to built-in types and removing duplicate built-in types after the first instance of them. This defines a priority list of acceptable built-in types for the result. Because a TYPEDEF union is unchanging within an ADM (see Section 3.4.2) a processor MAY cache this flattened type list.

2. The input value is first matched against each type in the list, in order. If any type does match the value, the input is taken as the result value. If no types directly match, the procedure continues to the next step.

3. The input value is attempted to be converted to each type in the list, in order. The first successful conversion is taken as the the result value. If none of the built-in types can successfully convert the input value, this procedure stops and is considered failed.

> | Because the processing in Steps 2 and 3 could successfully match multiple built-in types for an input value, the ordering of that list (and thus the ordering of the union members which flattened to produce that list) is significant.

6.12. Value Comparing

Comparing of values in the AMM is needed for explicit basic operators (Section 4.3.3) but equality comparison is also needed for intrinsic logic of keys within AM values, fields of EXECSET (specifically the nonce for Agent processing and Manager aggregation) and RPTSET (specifically for Agent aggregation and Manager correlating).

6.12.1. Equality and Inequality

Comparison of equality and inequality are about two values being exactly equal by-value. This does not include consideration of two values "meaning" the same thing (_e.g._ between a typed INT and REAL32 value) but there is an allowance in the following procedure for converting untyped literals. Users providing expressions to Agents need to consider this when comparison operators are included.

Comparison of two values for equality SHALL use the following rules:

* Two untyped literal values are equal if their primitive values are equal.

* Two typed literal values are equal if either both values have the same literal type and equal value.

* A typed and untyped literal are equal if the untyped value can be converted to the literal type of the other and those two values are equal.

* Two object reference values are equal if their path parts and their parameters are all equal by-value. This means that two references which refer to the same object using different combination of identifiers are not considered equal.

* Two namespace reference values are equal if their path parts are all equal by-value. This means that two references which refer to the same namespace using different combination of identifiers are not considered equal.

* All other combinations are not equal (_e.g._, between a literal and a namespace reference value).

  | URI references do not technically exist within the AMM value
  | system, they are a side effect of ARIs within an ADM definition
  | and rely on the ARI Resolving procedure to obtain absolute
  | object or namespace reference values.

6.13. Value Compacting

The procedures in this section allow AMM values, both typed literals and parameterized object references, to be compacted by eliding specific information based on the context of an associated semantic type or object formal parameters. The context for this compacting activity is taken from the model in which the specific values are being used.

When an object formal parameter, table template column, or report template item is known to have a semantic type that will match and convert from either a typed or an untyped literal with the same result value, the type of the literal can be elided without affecting the result. Another way of looking at this compacting is that when a literal value has the same type as its context requires, then the value’s type is redundant and can be elided without loss of information.

Even in the absence of an associated model, there are compacting rules that can be followed to elide unneeded information for specific built-in literal types. For example, the BOOL value true exists only within that type while the UINT value 5 is also within the domain of INT and several others.

6.13.1. Context Type

The context type for an object formal parameter (Section 3.4.1) SHALL be the type of that parameter. The context type for a table template column (Section 3.3.5) SHALL be the type of that column. The context type for a report template item (Section 4.2.4.3) which contains a VALUE-OBJ (Section 4.2.5) reference SHALL be the produced type of that referenced object. All other report template items, which must be EXPR literals, do not have an associated context type.

6.13.2. Literal Values

When compacting a literal value based on a context, the processor performs the following:

1. If the value has type NULL, BOOL, TEXTSTR, or BYTESTR the type is removed from the value and this procedure is finished. These types have unambiguous, unique value representation.

2. The associated context type is determined based on the rules of Section 6.13.1. If there is no such context type, this procedure is finished with the value unchanged in this step.

3. If removing the type from the value has no effect on how it would be converted (Section 6.11) to the context type, the type is removed from the value.

4. If the value is one of the containers (Section 3.2.2) and the context type is a semantic type for that container (_i.e._, not just the literal type AC, AM, or TBL), each of the constituent values is compacted based on the corresponding item of the context semantic type.

6.13.3. Object Reference Values

When compacting an object reference value based on a context object, the processor performs the following:

1. If the reference value contains given parameters in the form of a map and can be converted to a list with no change in the actual parameters (in accordance with Section 6.3.1), the parameters are converted to a list. Eliding the map keys will always compact the given parameters.

2. Each of the given parameter values is itself compacted based on the corresponding formal parameter context.

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.

7.1. CONST Definitions Have Life Cycles

All CONST objects SHOULD be given names and behaviors that reflect the requirements of Section 3.4.5, specifically that the stored and produced value cannot change after it is initially defined in a model. This does not, however, mean that a model needs to contain the same CONST definitions for its entire lifetime.

When new CONST objects are needed they can be added to a model without any additional consideration, they will just be given a unique name and enumeration. When a CONST object needs to be superseded, the old CONST object can have its status changed to "deprecated" or "obsolete" and a new object defined to hold the new value (see Section 7.16). If this kind of CONST life cycle management is done in an ADM it would likely require software update on the Agent but if it is done in an ODM it can be done during normal continuous runtime.

7.2. VAR Definitions Need to Consider State Changes

All VAR objects SHOULD be given names and behaviors that reflect the requirements of Section 3.4.10, specifically that variables are completely Manager-controllable within their defined semantic type constraints (Section 3.3). Application behavior SHOULD be reasonable for any VAR value state that matches (Section 6.10) its associated semantic type. A "reasonable" application behavior does not need to be successful in its application-domain processing but it does need to accept the VAR state for such processing. Application behavior SHOULD be reasonable when responding to changes of VAR value state over time. A "reasonable" application can impose hysteresis timeouts to avoid excessive processing if the state of a VAR changes frequently.

7.3. EDD Definitions Need Nilpotency

All EDD objects SHOULD be given names and behaviors that reflect the nilpotency requirements of Section 3.4.4. Agent behavior SHOULD be reasonable even if duplicate and concurrent EDD value production is performed.

7.4. CTRL Definitions Need Idempotency

All CTRL objects SHOULD be given names and behaviors that reflect the idempotency requirements of Section 3.4.6. For example the term "ensure" is preferable to "add or modify". Likewise "discard" is encouraged instead of "remove if necessary".

Agent behavior SHOULD be reasonable even if duplicate and concurrent CTRL executions are performed. Consider an "add-" CTRL that fails if a value already exists as opposed to an "ensure-" CTRL that checks a precondition and stops, thus guaranteeing idempotency.

7.5. OPER Definitions Need Nilpotency

All OPER objects SHOULD be given names and behaviors that reflect the nilpotency requirements of Section 3.4.7. For example, evaluating an OPER cannot have visible side effects beyond trivial activities such as logging or accounting. Agent behavior SHOULD be reasonable even when concurrent OPER evaluation is performed.

7.6. Choosing between VAR or Application-Managed State

Within an ADM, configuration state can be managed in-Agent using one or more VAR objects or can be manged out-of-Agent in the application itself, using CTRL and EDD objects to manipulate and introspect the state respectively. The choice of which to use depends on the kind of data being managed and how that data can or needs to be controlled.

VAR-Managed State: In this case the entire state is managed by VAR objects inside an Agent, where each object keeps an individual atomic portion of the state. Common CTRLs to inspect, report on, and manipulate VAR state are provided by the Agent ADM. No other Manager-visible objects are necessary to support this case.

A precondition to managing state entirely within VAR objects is that the state allocated to each VAR is logically atomic and independent of any other VAR or application state. The major benefit to this case is that all of the bookkeeping and persisting of state is performed within the Agent proper and the application is able to access, and possibly modify, that state.

An example of a VAR-managed state is the administrative state (enabled or disabled) of a protocol endpoint, where a Manager has full control over the state and can change it at-will. The application will react to changes of the VAR state to perform subsequent actions or other application-internal state changes. In this example, there would likely be a corresponding EDD-exposed operational state and possibly usage state to match the logical models of [RFC4268] and [X.731].

Application-Managed State: In this case the state exists outside the Agent and is accessed by the Manger, through the Agent, via EDD objects used to expose the state (or subsets of the state) and CTRL objects used to directly manipulate or indirectly affect the state. Those CTRL objects are necessarily application-specific and state-specific.

This case does not have any preconditions so it is general-purpose, but it does impose a burden on the application side to define those interface objects and to implement the Agent-facing behavior associated with those objects. A major benefit to this case is that it is completely up to the application to determine how and when the state can change. This allows an application to enforce internal consistency across different portions of state as necessary.

An example of an application-managed state is a large table of items, where the table is exposed by a single EDD along with a CTRL to ensure that specific rows are present and CTRL to discard specific rows (by either some unique ID value or a row-matching expression). Keeping the entire table in the application instead of a direct VAR state avoids the need to replace the entire table each time an incremental update is needed.

7.7. Choosing between CONST, VAR, or EDD

The purpose of associated data and expected behavior of values over time are the main influences of which value-producing object type to use for different purposes.

CONST Objects: These are recommended for data which are expected to have a long use lifetime and in situations where synchronization between Agent and Manager is difficult or impossible. When storing RPTT or MAC values on an Agent, using a CONST object ensures that any time the value is needed within an execution the object reference will produce the same known value. For example, any time a report is received with a source referencing a CONST object the associated RPTT can only have one possible content (meaning one possible interpretation of the report items). When a

new value is needed in this case, the old CONST object can have its status changed to "deprecated" or "obsolete" and a new object defined to hold the new value.

VAR Objects: These are recommended for data which are expected to have shorter lifetime or need to be influenced by self-modifying logic on the Agent itself. When storing RPTT or MAC values in a VAR object, it requires some synchronization between Agent and Manager. For example, any time a report is received with a source referencing a VAR object the Manager needs precise state history to know what the stored RPTT value of the VAR was when the report was generated. The benefit of using a VAR in this case is that a single object can be used to store a time-varying value (in either an ADM or an ODM with no change in behavior). VAR objects are also useful for storing threshold or limit values for state-based rules and future in-Agent alarm-condition logic.

EDD Objects: These are necessary when data originates from an application outside the Agent (as its name implies) but can also be used when it is more convenient to have application-controlled, possibly complex, values than try to manage the state of a VAR object. It is perfectly allowed that a dynamically constructed RPTT value is produced by an EDD and used for reporting. Doing so just requires the Manager receiving the report to be able to have precise information about what the RPTT content would have been at the time of reporting. EDD objects can also be coupled with one or more CTRL objects to enforce more complex state limitations than one an individual VAR with a single semantic type (see Section 7.2).

For example, a set of inter-related EDDs can be defined to allow introspecting the internal state of some application algorithm while an associated parameterized CTRL object is used to alter the configuration of the algorithm, which then alters the state, and finally is reflected in changes to the EDD values. Rather than attempting to treat the configuration inputs as independently changing VAR objects, the CTRL parameters are updated together by the application so that the entire set of configuration is either considered valid or not. If inputs were taken from separate VAR objects, there are likely to be intermediate states when some (but not all) VAR states are altered and the whole ensemble is invalid. By using the CTRL parameters as input, the modification becomes atomic from the Manager’s perspective when the CTRL execution finishes.

7.8. Choosing a Semantic Enumeration or an IDENT Hierarchy

The choice between using a static definition of enumerated values within a Named Type Use (constraining an integer built-in type) or using an extensible IDENT object (Section 3.4.3) hierarchy depends on several factors.

Static or Extensible: This consideration is about how many enumerated values are currently known and how much it is expected to grow in the future. A semantic enumeration is useful when the set of values is small, well-known, and not likely to be expanded upon. An IDENT hierarchy is more useful when an initial set of values is known but expected to grow in unknown ways in the future.

Localized or Distributed: This consideration is about how localized the choices are; whether or not a single model can define all options or whether some options need to be defined across multiple models. An IDENT hierarchy allows definitions of choices to span across multiple models and be defined at different times. It also allows some choices to be made optional by being present in models which will not be implemented in some Agents. Knowing which models (and which features) are implemented in an Agent would determine which IDENT choices are available.

Parameterized options: This consideration is whether or not there are secondary options related to each enumerated value. An IDENT hierarchy allows object-specific parameters to be associated with all non-abstract IDENTs (see Section 7.12) which allows a form of type safety for IDENT-enumerated choices with parameters. A static enumeration of choices does not allow for per-choice parameters.

For example, a model which needs to identify a unidirectional link as either being outgoing from or incoming to a node can simply use a static enumeration of two possible values. The value itself then takes the form of one of the enumerated integer choices with a specific human-friendly name associated with each one, and the encoding will be very concise. The options in this case are limited by design and cannot be expanded upon without changing the meaning of the type.

In another example, a model needs to make use of a transport endpoint address but does not want to be limited to a single possible transport protocol or addressing scheme. In this case, the model can define (or import) an abstract base IDENT object and use a value type of an IDENT derived from that base object. The value then takes the form of an object reference (to a non-abstract, derived IDENT), this

is less concise than a single integer but is completely open-ended for possible future support. The model can itself define some specific derived IDENT objects or defer that to other, transport-related models.

7.9. Choosing a Table Template or a Uniform List

The choice between using a Table Template or a Uniform List of possibly complex structures depends on several factors.

Static set of Fields: If the number of fields and and type of each field of a structure is known at design time, a table template is a simpler and easier to understand by mapping each field to a table column. The table template is restrictive, similar to the semantic enumeration, in that the columns of a single template are static and cannot be expanded upon (without creating a new semantic type, see Section 7.16).

Row Variability: If there is only a single field or a type-variable field which is more suitable to a parameterized IDENT (Section 7.12) then a uniform list is more suitable. This especially applies in a situation where the items of the list need to be extensible, which the IDENT hierarchy can handle easily. The drawback to using an IDENT hierarchy for each item is the complexity of specifying the desired schema and for users to understand how the model works.

For example, if an application manages a list of entities, each with uniform fields and types, those fields can be converted into columns of a table template and the state of those entities can be encoded all together as a table value.

7.10. Use Tables for Related Data

In cases where multiple EDD or VAR values are likely to be produced and evaluated together, that information SHOULD be placed in an Table Template (Section 3.3.5) semantic type within a single EDD or VAR rather than defining multiple independent EDD and/or VAR objects. By making a Table Template, the relationships among 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 increased transport and processor utilization and the potential for non-synchronized access across multiple value productions or when modifying VAR state (see Section 7.2).

7.11. Use Common TYPEDEFs and Semantic Types

Even when values in EDDs or VARs are simple integer counters or gauges, it is strongly RECOMMENDED for ADM authors to make use of semantic types in pre-existing TYPEDEF objects (within shared-use ADMs). This is preferred over either creating their own semantic types at the point of use or using simple built-in types without any documented semantics. Both of those alternatives are worse for maintenance and likely to be confusing to users of the ADM, both on the Manager side and the Agent implementation side.

7.12. IDENT Objects with Parameters

For simple uses of an IDENT hierarchy, the name and enumeration of each non-abstract IDENT function as a sort of extensible choice, where each non-abstract object represents one possible choice and a reference to that object is the AMM value for the choice. This is a valuable use of an IDENT hierarchy but not the only possible one. Because IDENT objects can each define formal parameters and IDENT references can use those with given parameters, it is possible to use non-abstract IDENT object to function as data structures.

A trivial way to use parameterized IDENT objects is to define an IDENT with parameters but without any base objects. This enables a single AMM value (from any value-producing object (Section 2.1) or within an expression) to contain an IDENT reference which itself contains given parameters associated with the object.

A more complex way to use parameterized IDENT objects is to construct a hierarchy using an abstract base IDENT (which itself cannot be parameterized), with any number of intermediate abstract derived IDENTs, and finally a set of non-abstract derived IDENTs having form parameters specific to that object’s meaning as a choice.

For example, an abstract "base-transport" object defined in a base model could have non-abstract derived IDENTS, each defined in a separate layer-specific model, including the following:

* An "ethernet-transport" with parameters of an Ethernet MAC address
* An "ip-transport" with parameters of an IP address and port number
* A "bpv7-transport" with parameters of an Endpoint ID URI

Because each of those parameters it itself typed and type-checked, this kind of structure allows a Manager to perform type checking before allowing values to be sent to an Agent, and allows a general-purpose Agent to validate values before they are passed up to an application or used to change the state of a VAR.

7.13. CONST and VAR Objects with Parameters

Unlike the behavior of parameterized EDD objects, for which parameters can be used to arbitrarily affect produced values, the value production procedures for CONST and VAR objects (defined in Section 6.5.1) allow parameters to "flow down" into stored values containing LABEL parameters. This allows things like CONST report templates to contain simple parameters to avoid the need to define several different, but mostly similar, report template values across multiple objects.

7.14. EDD Objects with Parameters

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.

Parameters incur transport and processing costs (see Section 6.3.1) 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 non-parameterized 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 Uses	Non-Parameterized Uses
./EDD/num_bytes_by_pri(low)	./EDD/num_bytes_by_low_pri
./EDD/num_bytes_by_pri(med)	./EDD/num_bytes_by_med_pri
./EDD/num_bytes_by_pri(high)	./EDD/num_bytes_by_high_pri

Table 6: Example Parameterized EDDs

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

7.15. OPER Objects with Parameters

For common uses of OPER objects, the functional behavior performed by the OPER requires only its operands to produce a result. For complex cases, though, an OPER can be defined to have formal parameters which influence its functional behavior.

Giving parameters to an OPER is a trade between less complex individual OPER definitions and consolidating behavior into a single parameterized definition. Parameters can increase expressiveness of a small number of objects, but can also lead to confusing semantics or overloaded objects which could produce the same behaviors by distributing over several, more specialized OPER objects.

One example is an OPER which has a parameter determining the number of operands to sum together. While the agent default "numeric addition" OPER is purely a binary function, a general purpose summation OPER can be defined which takes a parameter to determine the number of operands to pop from the evaluation stack and sum together.

Another example is an OPER which has parameter(s) defining algorithmic configuration separate from the operand(s) which are treated as algorithmic input. The key distinction is that the given parameter values are treated as constants from the perspective of the expression evaluation and _cannot_ be taken from the evaluation stack.

7.16. Object Life Cycle

Within a single model (whether ADM or ODM) each object has a well-defined life cycle described below.

1. At a specific model revision an object comes into existence with a "current" status.

2. At a later revision an object can change to "deprecated" status, which means it continues to exist but hints to users that this will change in the future.

3. Finally, a later revision the object will change to "obsolete" status and become unusable, the model definition will continue to exist for historical reference to a user but it has no further meaning to an Agent.

This single-object life cycle can be combined across multiple objects to support transitioning of behavior or structure across revisions of a single model. For example, an outdated TYPEDEF or IDENT object can be deprecated in the same revision that a replacement object is defined (with a descriptive note in the outdated object pointing to its replacement) and later on the outdated object can be made obsolete and no longer usable. This behavior enables some level of adaptation and growth in a model without need to use more complex mechanisms of model augmentation, where being able to interpret a single structure requires information across several different models.

7.17. Application State Visibility

When configuration or state is managed using VAR objects, within either ADMs or ODMs, there is a built-in Agent ability to both alter and inspect the stored value of the object. This provides a large benefit to the Manager side for simple activities such as testing or diagnostic behaviors in Applications which make use of the VAR values.

Unlike the use of VAR objects, which are managed by the Agent, CTRLs will typically have side effects in application state which are outside of the Agent proper. When an ADM provides CTRL objects for changing application configuration or state, it is RECOMMENDED that the same ADM also provide EDD objects for inspecting the resulting application state. This introspection ability might not be needed operationally, but for testing and diagnostics it is important.

8. IANA Considerations

This document does not make any changes to IANA registries.

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.

The Access Control Lists (ACLs) functionality presented in this document would be implemented separately from network security mechanisms.

ACL groups are expected to be associated with Managers. However, the form of Manager identification must be provided by separate transport-specific ADMs. The AMM provides no general purpose identifier, such as peer name and address, that would be required to uniquely describe each Manager.

10. References

10.1. Normative References

[IEEE.754-2019]
IEEE, "IEEE Standard for Floating-Point Arithmetic", IEEE IEEE 754-2019, DOI 10.1109/IEEESTD.2019.8766229, 18 July 2019, <https://ieeexplore.ieee.org/document/8766229>.

[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>.

[RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, DOI 10.17487/RFC3629, November 2003, <https://www.rfc-editor.org/info/rfc3629>.

[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>.

[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>.

[RFC7950] Bjorklund, M., Ed., "The YANG 1.1 Data Modeling Language", RFC 7950, DOI 10.17487/RFC7950, August 2016, <https://www.rfc-editor.org/info/rfc7950>.

[RFC8341] Bierman, A. and M. Bjorklund, "Network Configuration Access Control Model", STD 91, RFC 8341, DOI 10.17487/RFC8341, March 2018, <https://www.rfc-editor.org/info/rfc8341>.

10.2. Informative References

[ECMA-262] Ecma International, "ECMA-262 12th Edition, June 2021. ECMAScript 2021 language specification", June 2021, <https://262.ecma-international.org/12.0/>.

[IANA-DTNMA] IANA, "Delay-Tolerant Networking Management Architecture (DTNMA) Parameters", <https://www.iana.org/assignments/TBA/>.

[RFC2578] McCloghrie, K., Ed., Perkins, D., Ed., and J. Schoenwaelder, Ed., "Structure of Management Information Version 2 (SMIv2)", STD 58, RFC 2578, DOI 10.17487/RFC2578, April 1999, <https://www.rfc-editor.org/info/rfc2578>.

[RFC2580] McCloghrie, K., Ed., Perkins, D., Ed., and J. Schoenwaelder, Ed., "Conformance Statements for SMIv2", STD 58, RFC 2580, DOI 10.17487/RFC2580, April 1999, <https://www.rfc-editor.org/info/rfc2580>.

[RFC4268] Chisholm, S. and D. Perkins, "Entity State MIB", RFC 4268, DOI 10.17487/RFC4268, November 2005, <https://www.rfc-editor.org/info/rfc4268>.

[RFC6991] Schoenwaelder, J., Ed., "Common YANG Data Types", RFC 6991, DOI 10.17487/RFC6991, July 2013, <https://www.rfc-editor.org/info/rfc6991>.

[RFC7942] Sheffer, Y. and A. Farrel, "Improving Awareness of Running Code: The Implementation Status Section", BCP 205, RFC 7942, DOI 10.17487/RFC7942, July 2016, <https://www.rfc-editor.org/info/rfc7942>.

[RFC8610] Birkholz, H., Vigano, C., and C. Bormann, "Concise Data Definition Language (CDDL): A Notational Convention to Express Concise Binary Object Representation (CBOR) and JSON Data Structures", RFC 8610, DOI 10.17487/RFC8610, June 2019, <https://www.rfc-editor.org/info/rfc8610>.

[RFC9675] Birrane, III, E., Heiner, S., and E. Annis, "Delay-Tolerant Networking Management Architecture (DTNMA)", RFC 9675, DOI 10.17487/RFC9675, November 2024, <https://www.rfc-editor.org/info/rfc9675>.

[I-D.ietf-dtn-ari] Birrane, E. J., Annis, E., and B. Sipos, "DTNMA Application Resource Identifier (ARI)", Work in Progress, Internet-Draft, draft-ietf-dtn-ari-07, 16 October 2025, <https://datatracker.ietf.org/doc/html/draft-ietf-dtn-ari-07>.

[I-D.ietf-dtn-amp] Birrane, E. J. and B. Sipos, "DTNMA Asynchronous Management Protocol (AMP)", Work in Progress, Internet-Draft, draft-ietf-dtn-amp-03, 15 January 2026, <https://datatracker.ietf.org/doc/html/draft-ietf-dtn-amp-03>.

[X.731] ITU-T, "Information technology - Open Systems Interconnection - Systems management: State management function", ITU-T Recommendation X.731, ISO/IEC 10164-2:1992, January 1992, <https://www.itu.int/rec/T-REC-X.731-199201-I/en>.

[github-dtnma-tools] JHU/APL, "A reference implementation of the DTN Management Architecture (DTNMA) Agent and related Tools", <https://github.com/JHUAPL-DTNMA/dtnma-ace>.

Appendix A. Access Control Lists

This section presents an overview of fine-grained application security using an Access Control List (ACL) within an Agent. The ACL itself is managed through a standard ACL ADM.

A table of ACL entries associating access permissions with groups of Managers and groups of objects is queried at runtime to ensure privileged access, while simultaneously allowing efficient implementation on an embedded device.

The concepts presented are in agreement with the Network Configuration Access Control Model (NACM) documented in [RFC8341].

A.1. Access Points and Permissions

The access control for an Agent is handled as a decision to allow or deny specific actions at specific points during the processing activities of an Agent and its applications. Some of these access points are enforced during control execution, either by controls within the Agent (provided by the Agent ADM) or controls defined by applications above the Agent. Other access points are part of the Agent-internal procedures themselves (see Section 6).

An access permission is an object within AMM the data model representing each access point. This allows the syntax of access control to use AMM values (object references) and types within the ACL ADM. While the ACL ADM defines several built-in permissions, the ACL is extensible and allows for fine-grained application-level access control managed through the same ACL.

Permissions are defined by ADMs by deriving an IDENT object from a specific abstract base object from the ACL ADM. Each permission object represents a single category of access enforced on a manager by a control execution, meaning outside the Agent at the application level, or inside an Agent-internal procedure. These access permissions are separate from and independent of any transport-layer access control which would affect how messages are able to be sent to or from an Agent.

Non-abstract permission objects can define formal parameters to add additional permission-specific narrowing of allowed access. Parameters on a ACL permission object are defined and handled just the same as for any other managed object.

A.2. Access Groups

Access groups provide a way to combine access control logic that reduces configuration and enables adaption of control to different sets of Managers over time. A group could be used as a "role," but is not limited by the Agent to have any specific semantic meaning. The ACL ADM contains EDDs for visibility into the defined groups as well as controls to allow management of those groups.

Each group is defined by a unique numeric identifier and a unique informative text name. The identifier is used for all cross-referencing from the ACL, while the name is more human-friendly and available for diagnostics.

The Agent itself is always the exclusive member of a built-in group number zero. This allows access control to be enforced for things like rule-triggered executions, or evaluation of the actual SBR conditions defining those rules. This does enable the possibility of a Manager elevating their access to the agent group by creating new rules, which highlights the importance of controlling the ability of a Manager to create rules in the first place.

Each manager can have zero or more access group associations, which is based on the transport endpoint identifier associated with that manager. These identifiers are transport-mechanism-specific, which is why the access group is a useful means to isolate ACL from the transport mechanisms available to an Agent.

A.3. Access Control List

The ACL itself is the way that combinations of access groups and object reference patterns are combined into a set of permissions to apply to each controlled access point. The ACL ADM contains EDDs for visibility into the ACL as well as controls to allow management of the ACL.

Access control permissions will be assigned by an ACL entry using the combination of: an access group, an object reference pattern (Section 3.2.3), and a set of (possibly parameterized) permission references. At runtime the Agent will retain a table of these tuples to store its access control configuration. The table will be queried by the Agent to find the corresponding permissions for each managed object at its time of access.

> | NOTE: There are optimizations an implementation could do to avoid time-consuming table lookups, because an ACL will change infrequently and objects are added to and removed from an Agent at controlled points. For example, when an ADM implementation is loaded or when an ODM is modified. Internally, the agent could associated access controls with an object and update object state as an ACL changes or when ACLs are being added.

Each agent activity will use a set of Groups corresponding to the appropriate execution context when querying for permissions. For more information see Appendix A.4.

An object’s ARI shall be used by the Agent when querying the table. An important consideration is that an ARI contains an organization ID, model ID, and object ID, each of which may be expressed in either a text or integer form. When looking up permissions the Agent needs to use all forms of identifier of the object itself - which can match all 8 possible combinations - and NOT limit lookup to a particular ARI used by the Manager to reference an object.

If the Agent discovers multiple tuples that correspond to an object, the AT with least permission should be applied to the object. For example, if three tuples would allow an operation and the fourth would not, the Agent should deny permission.

Permissions shall be loaded during agent initialization and may be changed by an operator with sufficient permission. The default, empty ACL causes all access to be denied.

A.4. Access Context

Each Agent procedure (_i.e._, execution, evaluation, reporting) will have an access context associated with a set of access groups. For direct execution and evaluation, caused by an EXECSET, the group will correspond to the Manager that requested the execution or evaluation to occur. For delayed execution, such as for a rule action, the execution context will refer to the built-in group for the Agent itself. When creating reports the access for producing or evaluating each report item is in the context of access groups for the Manager receiving the report.

A.5. Enforcement

Access control shall be enforced in the following way for processing activities described in Section 6. The simple fact of referencing an object is not enforced by this model in any way, it is only when an object reference is acted upon at an access point within an activity that the ACL comes into effect.

The access points defined within the AMM for the base Agent activities and Agent ADM are listed below along with some examples of activities which are not access points.

ARI Resolving and Object Dereferencing: There is no explicit permission to control resolving of or dereferencing of particular object references or matching object reference patterns.

Execution: A single non-parameterized "execute" permission is

defined to allow value execution of associated CTRL objects. Although the permission is not parameterized, it will affect the ability to execute controls which are or are not parameterized equally.

Value Production: A single non-parameterized "produce" permission is defined to allow value production for associated objects. This permission is meaningful only for CONST, VAR, and EDD object types.

Evaluation: There is no explicit evaluation permission, either for expressions or individual OPER objects. Because the first step of evaluation is to expand object references into produced values, the value production permission affects evaluation indirectly. When production is denied, the expression will see an undefined value.

Reporting: There is also no explicit reporting permission. The value production permission affects the visibility of RPTT values produced by some objects and the contents of individual reporting items. When production is denied for a RPTT-producing object, the reporting will fail When production is denied for a report template item, the associated report item will have an undefined value.

Namespace (ODM) Creating and Obsoleting: A pair of non-parameterized "create-odm" and "delete-odm" permissions are defined to allow all namespace-manipulation controls in the Agent ADM (Section 4.3.1) to share common logic for creating and obsoleting whole ODMs.

Object Creating and Obsoleting: A pair of non-parameterized "create-object" and "delete-object" permissions are defined to allow all object-manipulation controls in the Agent ADM (Section 4.3.1) to share common logic for creating objects and obsoleting objects within an ODM. This permission applies during execution of those controls separately from permission to execute those controls at all.

Variable State Changes: Similarly, a single non-parameterized "modify-var" permission is defined to allow variable-state-changing controls in the Agent ADM (Section 4.3.1) to share common logic. This permission applies during execution of those controls separately from permission to execute those controls at all.

Implementation Status

This section is to be removed before publishing as an RFC.

[NOTE to the RFC Editor: please remove this section before publication, as well as the reference to [RFC7942], [github-dtnma-tools].]

This section records the status of known implementations of the protocol defined by this specification at the time of posting of this Internet-Draft, and is based on a proposal described in [RFC7942]. The description of implementations in this section is intended to assist the IETF in its decision processes in progressing drafts to RFCs. Please note that the listing of any individual implementation here does not imply endorsement by the IETF. Furthermore, no effort has been spent to verify the information presented here that was supplied by IETF contributors. This is not intended as, and must not be construed to be, a catalog of available implementations or their features. Readers are advised to note that other implementations can exist.

An implementation in C11 language of the procedures from this AMM is present in the reference DA (REFDA) library and example executables built from the apl-fy24 development branch of [github-dtnma-tools]. This repository includes unit test vectors for verifying DA behavior under many of the procedures of Section 6.

Acknowledgments

The following participants contributed technical material, use cases, and useful thoughts on the overall approach captured in this document: David Linko, Sarah Heiner, and Jenny Cao of the Johns Hopkins University Applied Physics Laboratory.

Authors' Addresses

Edward J. Birrane, III
The Johns Hopkins University Applied Physics Laboratory
11100 Johns Hopkins Rd.
Laurel, MD 20723
United States of America
Phone: +1 443 778 7423
Email: Edward.Birrane@jhuapl.edu

Brian Sipos
The Johns Hopkins University Applied Physics Laboratory
Email: brian.sipos+ietf@gmail.com

Justin Ethier
The Johns Hopkins University Applied Physics Laboratory

Email: Justin.Ethier@jhuapl.edu