Automated Infrastructure Management (AIM) Systems

Network Infrastructure Modeling and Systems Integration

Following the ISO/IEC 18598/DIS Standards Specification

Mihaela Iridon
Cândea LLC for CommScope, Inc.
Dallas, TX, USA
e-mail: iridon.mihaela@gmail.com

Abstract— **Automated Infrastructure Management (AIM) systems are enterprise systems that provision a large number and variety of network infrastructure resources, including premises, organizational entities, and most importantly, all the telecommunication and connectivity assets. In 2016 the International Standards Organization released the ISO/IEC 18598 specifications that provide standardization and sensible guidelines for exposing data and features of AIM systems in order to facilitate integration with these systems. CommScope, the primary contributor in defining these standards, has implemented these specifications for their imVision system [1]. This paper elaborates primarily on the ISO-recommended infrastructure elements and how to design the resource models that represent them. It also discusses the layered architecture used to build CommScope's imVision AIM system, and briefly describes a possible integration scenario between two AIM systems. Additionally, this article intends to share design and technology-specific considerations, challenges, and solutions adopted by CommScope, so that they may be translated and implemented by other organizations that intend to build - or integrate with - an AIM system in general.**

Keywords-automated infrastructure management (AIM); system modeling; network infrastructure provisioning; data integration.

I. Introduction

Modeling network infrastructure elements (ports, modules, patch panels, servers, cables, circuits, etc.) and building effective network management systems is a rather challenging task due to the complexity and large variety of telecommunication assets [1] and vendor implementations. Such systems are also designed to model and automatically detect physical connectivity changes and manage infrastructure and data exchange with other systems.

Until recently, no common representation of such elements existed, so that network infrastructure management providers defined their own proprietary models. For this reason, the task of integrating with these systems posed a high degree of complexity, forcing integrators to define highly specialized solutions and models, and potentially unwieldy model and data transformations to enable compatibility between the integrating systems.

CommScope has identified the stringent need to create a unified representation of telecommunication assets to help build and integrate with Automated Infrastructure Management (AIM) systems and worked with the International Standards Organization towards achieving this goal. The result of this collaboration was the ISO/IEC 18598 standard [2], a set of guidelines for modeling and provisioning AIM systems. These specifications were captured and extended in [1] and are also the main focus of this paper, by including modeling details that bring more clarity, add context, and provide further guidelines to the information described in the standards document. Identifying and organizing AIM system's assets in a logical and structured fashion allows for an efficient access and management of all the resources administered by the system.

This paper is organized around six sections as follows.

Section II presents several resource models from the perspective of designing RESTful services [3] [4] [5], with focus on the telecommunication assets, as proposed and used by CommScope's imVision API. This section also presents a solution for handling a large variety of hardware devices while avoiding the need for an equally large number of URIs for accessing these resources.

Section III discusses system architecture, patterns and design-specific details, elaborating on a few practical challenges, followed by noteworthy technology and implementation aspects captured in Section IV.

Section V examines options for integrating two or more AIM systems, specifically two CommScope AIM systems: imVision and the Quareo Middleware API. A high-level solution employing a variant of the Normalizer integration pattern is presented along with a few data integration and data layer modeling objectives.

Finally, Section VI attempts to join and summarize the main ideas and analysis points presented in this paper.

II. AIM System Domain Analysis and Resource Modeling

As with every software system – and more so with enterprise-level applications – domain modeling is of vital importance as it helps define, organize, and understand the business domain, facilitating the translation of requirements into a suitable design [6]. However, dedicated models can and should be designed for the various layers of a system's architecture [7]. Defining clean boundaries between the system's domain and the integration models [8] [9] as well as ensuring the stability of these models (via versioning) are imperative requirements for building robust and extensible systems, while allowing the domain models – both structural and behavioral – to evolve independently [3] [10].

The specification of the AIM resource model described here employed various design and implementation paradigms. However, all concrete resource types exposed by the system are simple POCOs (Plain Old CLR Objects for the .NET platform) or POJOs (Plain Old Java Objects for the Java EE platform). These models represent merely *data containers* that do not encapsulate any behavior whatsoever. Functional attributes are specific to the physical entities being modeled and are exposed only from the perspective of the system's connectivity they describe. The purpose of the AIM model described here and in [1] is to define a common understanding of the data that can be exchanged with an AIM system while any specific behavior around these data elements is left to the implementation details of the particular system itself.

As opposed to the design principles of stateful services (such as SOAP and XML-RPC-based web services) – where functional features and processes take center stage while data contracts are just means to help model those processes [9] [7], in RESTful services the spotlight is distinctly set on the transport protocol and entities that characterize the business domain. These two elements follow the specifications of Level 0 and 1, respectively, of the RESTful maturity model [11] [5]. The resources modeled by a given system also define the service endpoints (or URIs), while the operations exposed by these services are simple, few, and standardized (i.e., the HTTP verbs required by Level 2: GET, POST, PUT, DELETE, etc.) [4] [5]. Nonetheless, in both cases, a sound design approach (as with any software design activity in general) is to remain technology-agnostic [6] [8] [7].

A. *Resource Categories Overview and Classification*

The entities proposed in the Standards document [1] are categorized by the sub-domain that they describe as well as their composability features. This classification helps define a model that aligns well with the concept of separation of concerns (SoC), allowing common features among similar entities to be shared effectively, with increased testability and reliability.

The ISO/IEC Standards document proposes the classification of resources shown in Table I. While some elements listed here may not be germane to all AIM systems, the Standards document intends to capture and categorize *all elements* that could be modeled by such a system.

TABLE I. RESOURCE CATEGORIES AND CONCRETE TYPES

PREMISES	Geographic Area, Zone, Campus, Building, Floor, Room
CONTAINERS	Cabinets, Racks, Frames
TELECOM ASSETS	Closures, Network Devices, Patch Panels, Modules, Ports, Cables, Cords
CONNECTIVITY ASSETS	Circuits, Connections
ORGANIZATIONAL	Organization, Cost Center, Department, Team, Person
NOTIFICATIONS	Event, Alarm
ACTIVITIES	Work Order, Work Order Task

It also proposes a common terminology for these categories so that from an integration perspective there is no ambiguity in terms of what these assets or entities represent, where they fit within an AIM system, and what their purpose is. It defines, at a high-level, the ubiquitous integration language by providing a clear description and classification of the main elements of an AIM system.

This paper analyzes these recommendations, materializes them into actual design artifacts – following the exact nomenclature used in the Standards document, and proposes a general-purpose layered architecture for the RESTful AIM API system while addressing a few concerns regarding AIM systems integration in general.

B. *Common Model Abstractions*

Since all resources share some basic properties, such as name, identifier, description, category, actual type (that identifies the physical hardware components associated with this resource instance), and parent ID, it is a natural choice to model these common details via basic inheritance, as shown in Figure 1. In order to support a variety of resource identifier types, e.g., Globally Unique Identifier (GUID), integer, string, etc., the `ResourceBase` class is modeled as a generic type with the resource and parent identifier parameterized by the generic type `TId`.

Of particular interest are *telecommunication assets* – the core entities in all AIM systems – a class of resource types which all must realize the `IAsset` marker interface – as proposed in [1] and in this paper.

C. *Designing the AIM Resource Models*

AIM systems are comprised of elements that fall into seven main categories. The modeling of these elements will be described following this standard classification which also aligns with the way these entities are organized into a compositional hierarchy; this grouping also defines the granularity and association relationship among them.

1) *Premise Elements*

A given organization's network infrastructure can be geographically distributed across multiple cities, campuses, and/or buildings, while being grouped under one or more sites – logical containers for everything that could host any type of infrastructure element. At the top of the infrastructure-modeling hierarchy, there are premises, which model location at various degrees of detail: from geographic areas and campuses to floors and rooms. Composition rules or restrictions for these elements may be modeled via generic type constraints, unless these rules are not enforced by a given system. Figure 2 shows the standards-defined premise entities, their primary properties, and the relationships between them.

2) *Telecom Connectivity Elements*

The main assets of a network infrastructure are its telecommunication resources, from container elements, such as racks and cabinets, to switches and servers, network-devices (e.g., computers, phones, printers, cameras, etc.), patch panels, modules, ports, and circuits that connect ports via cables and cords.

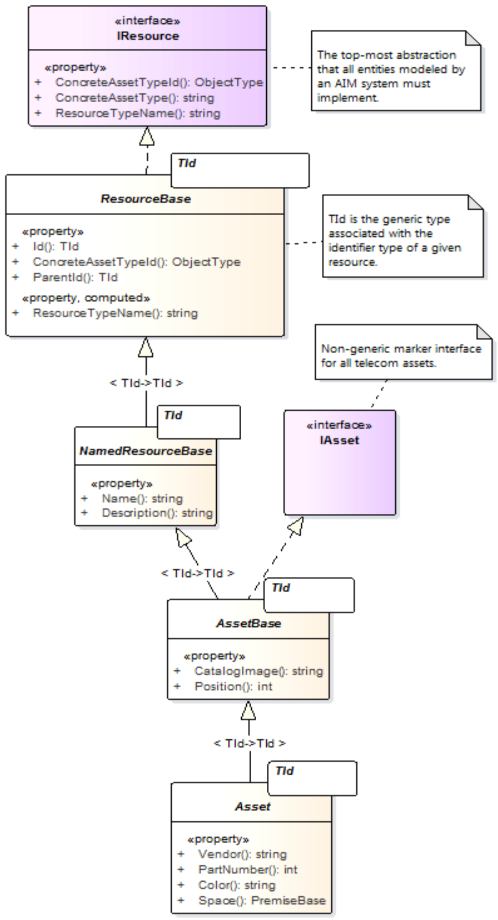


Figure 1. Resource Base Models

The diagram included in Figure 3 shows these asset categories modeled via inheritance, with all assets realizing the `IAsset` marker interface. As is the case for CommScope's imVision system, the type of the unique identifier for all resources is an integer; hence, all resource data types will be closing the generic type `TId` of the base class to `int: ResourceBase<int>`. This way, the RESTful API will expose these AIM Standards-compliant data types in a technology- and implementation-agnostic way that reflects the actual structure of the elements, while generics and inheritance remain transparent to integrators, regardless of the serialization format used (JSON, XML, SOAP). This fact is illustrated in Figure 5, which shows a sample Rack instance serialized using JSON. In addition to the elements shown in Figure 3 that support a persistent representation of the data center's telecom assets, there are those that describe the physical connectivity (i.e., circuits): cables, connectors, and cords. They play a chief role in defining the connectivity dynamics of the system. Figure 4 shows the primary resources for modeling this aspect of an AIM system.

3) *Organizational Elements*

Large AIM systems typically provision entities that describe the organization responsible for maintaining and administering the networking infrastructure. For example, tasks around the management of connectivity between panels and modules is usually represented by work orders and tasks which, in turn, are assigned to technicians.

4) *System Notifications and Human Activity Elements*

Hardware components of AIM systems, e.g., controllers, discoverable/intelligent patch panels and in some instances intelligent cords (e.g., CommScope's Quareo system) allow continuous/automatic synchronization of the hardware state with the logical representation of the hardware components.

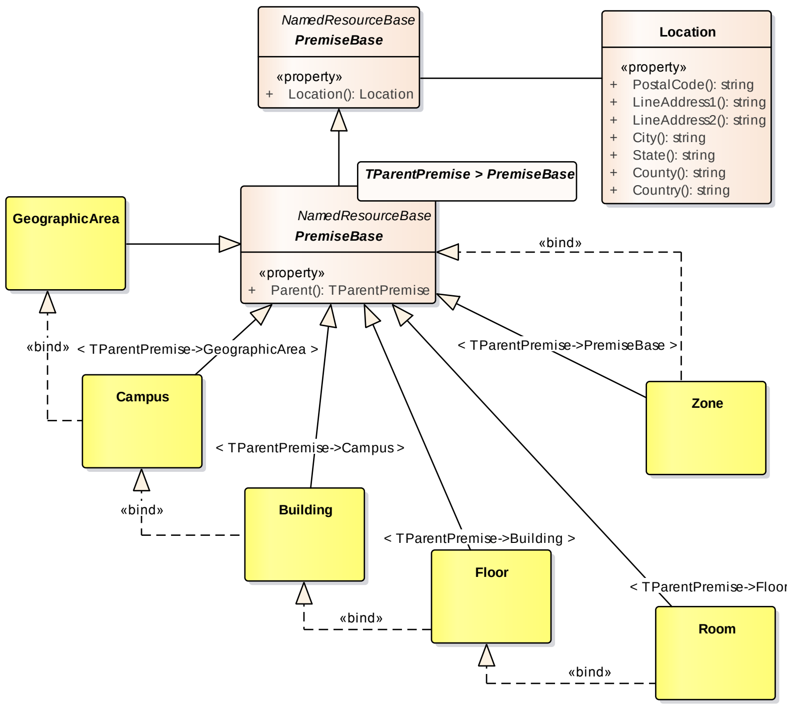


Figure 2. Premise Resource Models

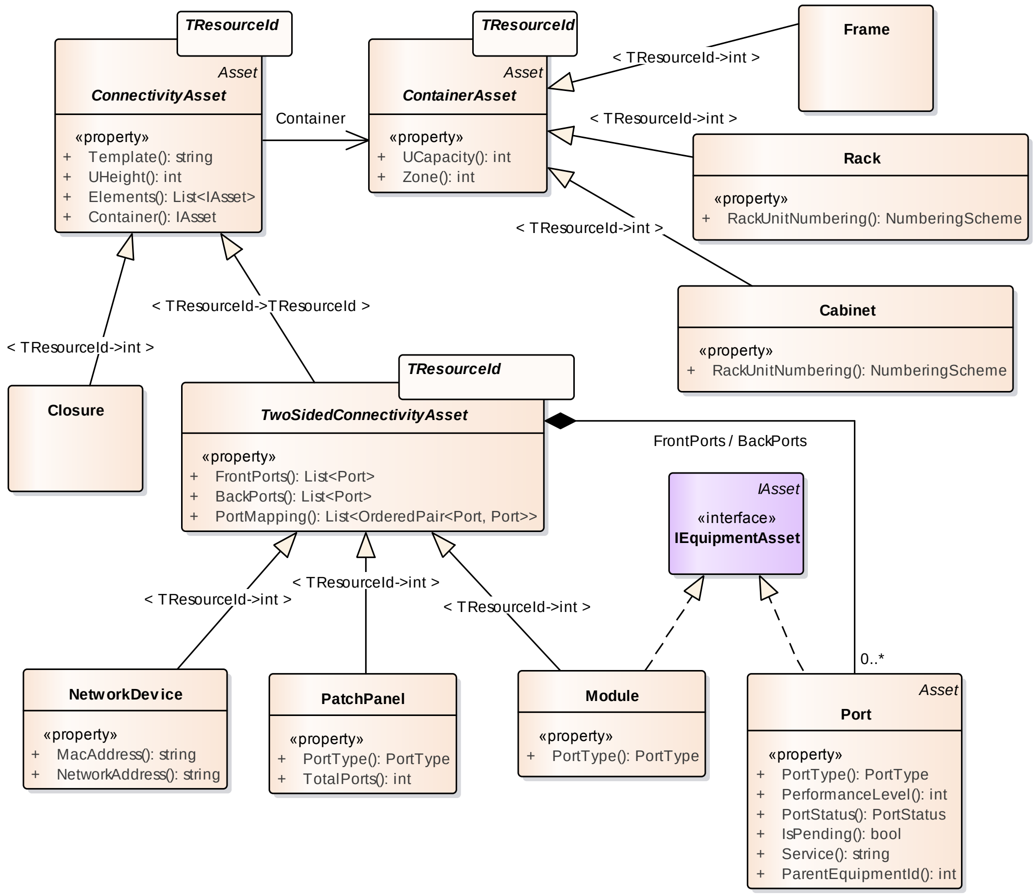


Figure 3. Telecommunication Assets Resource Models

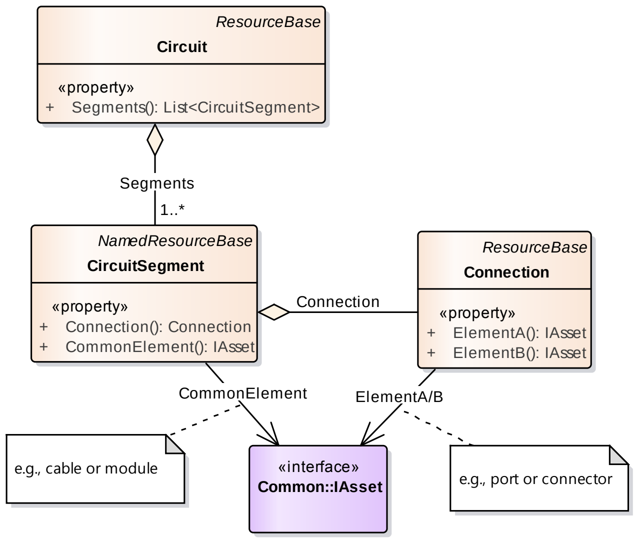
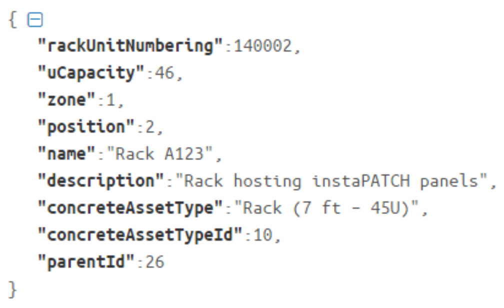


Figure 4. Connectivity Models



```
{ ⊟
   "rackUnitNumbering":140002,
   "uCapacity":46,
   "zone":1,
   "position":2,
   "name":"Rack A123",
   "description":"Rack hosting instaPATCH panels",
   "concreteAssetType":"Rack (7 ft - 45U)",
   "concreteAssetTypeId":10,
   "parentId":26
}
```

Figure 5. A JSON Representation of a Rack Resource

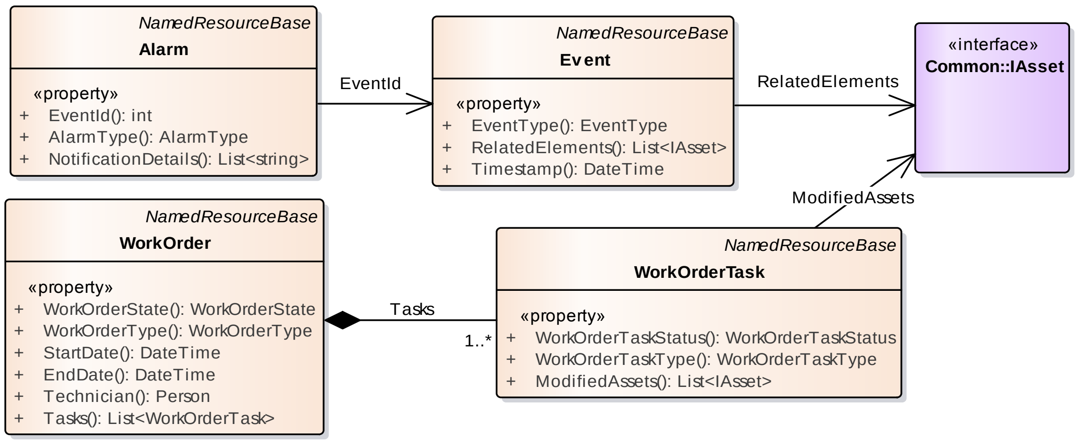


Figure 6. Notification and Activity Models

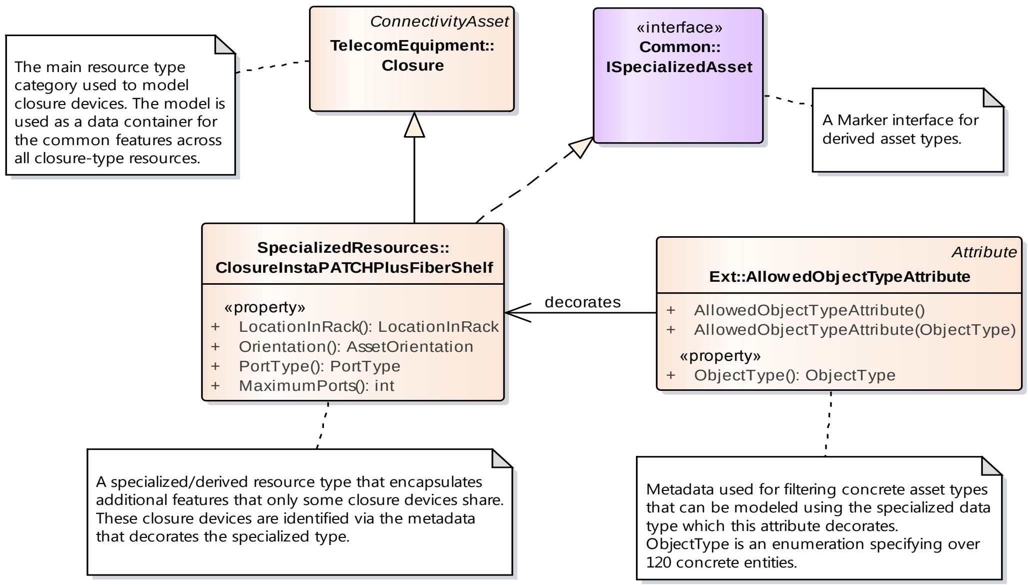

The main resource type category used to model closure devices. The model is used as a data container for the common features across all closure-type resources.
ConnectivityAsset
TelecomEquipment::
Closure
«interface»
Common::
ISpecializedAsset
A Marker interface for derived asset types.
SpecializedResources::
ClosureInstaPATCHPlusFiberShelf
«property»
+ LocationInRack(): LocationInRack
+ Orientation(): AssetOrientation
+ PortType(): PortType
+ MaximumPorts(): int
decorates
Attribute
Ext::AllowedObjectTypeAttribute
+ AllowedObjectTypeAttribute()
+ AllowedObjectTypeAttribute(ObjectType)
«property»
+ ObjectType(): ObjectType
A specialized/derived resource type that encapsulates additional features that only some closure devices share. These closure devices are identified via the metadata that decorates the specialized type.
Metadata used for filtering concrete asset types that can be modeled using the specialized data type which this attribute decorates.
ObjectType is an enumeration specifying over 120 concrete entities.

Figure 7. A Sample of a Specialized Closure with Additional Properties

This synchronization is facilitated by the concept of events and alarms that are first generated by controllers (**Alarm**) and then sent for processing by the management software (**Event**). These notification resource types are supported by the AIM Standards and are modeled as shown in Figure 6. This also includes activities that technicians must carry out, such as establishing connections between assets, activities that in turn trigger alarms and events, or are created as a reaction to system-generated events.

D. *Modeling Large Varieties of Hardware Devices*

The telecom asset model presented in Figure 3 depict the categories that define all or most physical devices seen in network infrastructure. However, actual hardware components have specialized features that are vendor-specific or that describe some essential functionality that the components provide. Such specialized attributes – like the ones shown in Figure 7 for a specific type of Closure – must be incorporated in the model for supporting the Add (POST) and Update (PUT) functionality of the RESTful services that expose these objects to the integrators. The main challenge then is: how to support such a large variety of hardware devices without having to expose too many different service endpoints, one for each of these specialized types?

According to the Richardson Maturity Model for REST APIs [11] – which breaks down the principal ingredients of a REST approach into three steps – Level 1 requires that the API be able to distinguish between different resources via URIs; i.e., for a given resource type there exists a distinct service endpoint to where HTTP requests are directed. For querying data using HTTP GET, we can easily envision a service endpoint for a given resource *category* – as per the models described above. For example, there will be one URI for modules, one for closures, one for patch panels, etc. However, when creating new assets, one must specify which *concrete* entity or device type should be created, and for this, the device-specific data must be provided. Since these supplementary features are not intrinsic to all objects that belong to that category, specialized models must be created – e.g., as *derived* types inheriting from the category models that encapsulate all relevant device-specific features.

For example, one of CommScope's connectivity products that falls under the category of Closures is the SYSTIMAX 360™ Ultra High Density Port Replication Fiber Shelf, 1U, with three InstaPATCH® 360 Ultra High Density Port Replication Modules [12] – a connectivity solution for high-density data centers that provides greater capacity in a smaller, more compact footprint. These closures come in a variety of configurations and aside from the common closure attributes (position, elements, capacity, etc.) other properties are relevant from a provisioning, connectivity, and circuit tracing perspective. Such properties include Orientation of the sub-modules, Location in Rack, Maximum Ports, and Port Type, as shown in the class diagram in Figure 7.

An alternative to using an inheritance model would be to create distinct types for each individual physical component that could be provisioned by the AIM system, but given the significant overlap of common features they can be consolidated and encapsulated in such a way that derived specialized models can be employed in order to increase code reusability, testability, and maintainability. The distinction between the various hardware components that map to the same specialized type can be managed, for example, via custom metadata associated with that data type (e.g., the **AllowedObjectTypeAttribute** in Figure 7).

E. Benefits of the Proposed Model

The models proposed in this paper are closely following the categories and elements outlined in the ISO/IEC standards. However, given the structural models presented here and taking advantage of certain technology-specific constructs and frameworks, there are some notable advantages resulting from the design of these models, related to their usage, and the integration capabilities for the services that expose them, with direct impact on performance, maintainability, testability, and extensibility.

✓ *Simplified URI scheme based on resource categories rather than specialized resource types*. This allows clients to access classes or categories of resources rather than having to be aware of - and invoke - a large number of URIs dictated by the large variety of hardware devices modeled. This also confers the API a high degree of stability, consistency, and extensibility even when the system is enhanced to provision new hardware devices.

✓ *Reduced chattiness between client application and services when querying resources* (**GET**). This benefit is directly related to the URI scheme mentioned above, since a single HTTP request can retrieve all resources of that type (applying the Liskov substitution principle [13]), even when multiple sub-types exist.

✓ *Reduced chattiness between client application and services when creating complex entities* (**POST**) *by supporting composite resources*. In some cases, the hardware device construction itself requires the API to support creating a resource along with its children in a single step (see Section IV.B for details). Child elements can be specified as part of the main resource or they can be omitted altogether while custom composition and validation frameworks resolve the missing sub-resources based on predefined rules.

Table II captures metrics regarding the request counts and sizes for creating a **PatchPanel** object.

✓ *Ample opportunity for automation when creating and validating composite resources*. Aside from considerably reducing the size of the request body given the option to omit child elements when adding new entities - as is the case for the imVision API – by employing frameworks that support metadata-driven automation, the API will ensure that the generated resource object reflects a valid hardware entity, with all the required sub-elements.

For the API consumers, this reduces the burden of knowing all the fine details about how these entities are composed and constructed. In some cases, the number of child elements to be created in the process depends on properties that the main resource may expose (e.g., **TotalPorts**) – which client applications will have to specify if the corresponding property is marked as [**Required**].

✓ *Extensible model as new hardware devices are introduced*. New models can easily be added to the existing specialized resources or as a new subtype. The interface for querying the data (**GET**) will not change. Adding/updating resources follows the Open/Closed principle [13] such that new types, properties, and rules can be added/extended without changing the already defined ones, thus ensuring contract stability.

TABLE II. POST REQUEST METRICS FOR QUATTRO PANEL (A PATCHPANEL RESOURCE)

Metric	Scenario	Value
Number of POST Requests	Without Support for Composite Resources	**31**: 1 for the Panel, 6 for the child Modules, and 6x4 for the ports
	With Support for Composite Resources	**1**: a single request for the Panel with its Modules (under **Elements**), with each Module being itself a composite resource containing 4 ports each, specified under the **FrontPorts** property of each Module
POST Request Body Size	With Explicit Children Included	21,449 bytes
	With No Children Specified (i.e., relying on the Framework to populate default elements)	572 bytes

III. A Proposed Layered Architecture for AIM API Integration Services

A. *Adding Integration Capabilities to an AIM System*

As per the Standards document guidelines [2], the AIM Systems should follow either an HTTP SOAP or a RESTful service design. Regardless of the service interface choice, there are several options for designing the overall AIM system. A common yet robust architectural style for software systems is the layered architecture [7] [8], which advocates a logical grouping of components into layers and ensuring that the communication between components is allowed only between adjacent or neighboring layers. Moreover, following SOLID design principles [13], this interaction takes place via interfaces, allowing for a loosely coupled system [14], easy to maintain, test, and extend. This will also enable the use of dependency injection (DI) or Inversion of Control (IoC) technologies such as Microsoft's Unity and MEF, or any other DI/IoC containers, to create a modular, testable, and coherent design [15].

CommScope's imVision system was built as a standalone web-based application, to be deployed at the customer's site, along with its own database and various middleware services that enable the communication between the hardware and the application. Relying on the current system's database, the RESTful Services were added as an integration point onto the existing system. The layered design of this new sub-system is shown in Figure 8 with the core component – the resource model discussed earlier – shown as part of the domain layer. The system also utilizes (to a limited extent) a few components from the legacy imVision system that encapsulate reusable logic. The diagram shows the actual design used for CommScope's imVision system.

Several framework components were used, most notably the *Validation* component, which contains the domain rules that specify the logic for creating and composing the various entities exposed by the API. These rules constitute the core module upon which the POST functionality relies. Along with the resource composition and validation engines, they constitute in fact a highly-specialized rule-based system that makes extensive use of several design and enterprise integration patterns that will be cataloged next.

B. *Patterns and Design Principles*

The various patterns and principles [8] [13] [14] employed throughout the design and implementation of the imVision API system are summarized in Table III. The automation capabilities built into imVision API mentioned earlier, that support creating composite object hierarchies, are a direct realization of the Content Enricher integration pattern used together with the Builder, Composite, and Specification software design patterns. From a messaging perspective, all requests are synchronous and only authorized users (Claim Check pattern) are allowed to access the API.

Design principles such as IoC/DI have been heavily used to deploy concrete implementation components (e.g., repositories, data access, etc.) to various layers of the application.

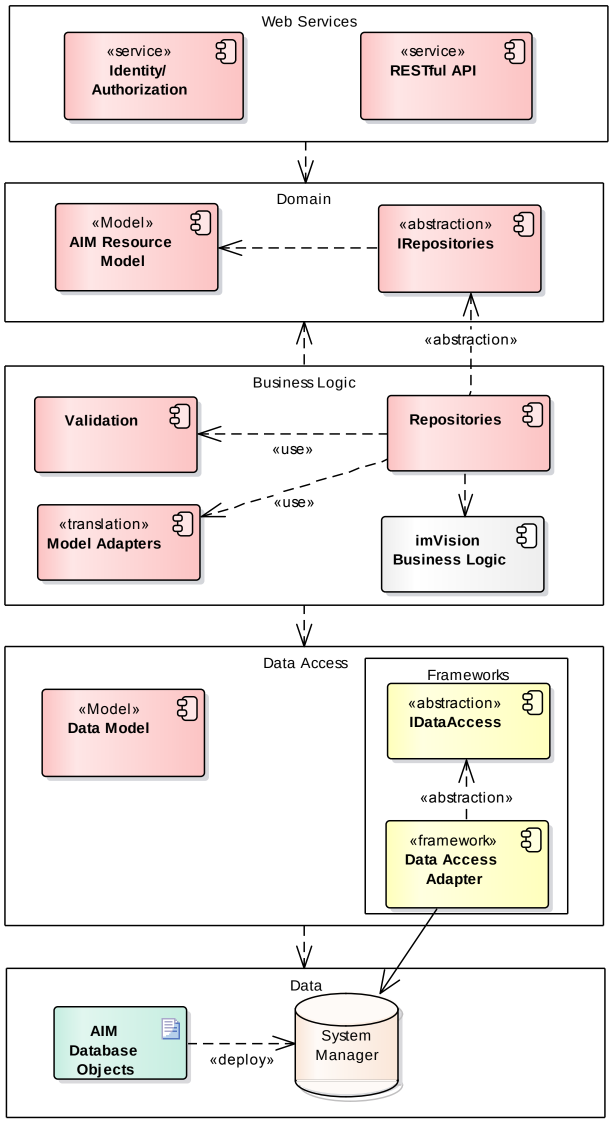


Figure 8. The Layered Architecture of the imVision AIM API

TABLE III. Design Patterns and Principles Employed

Design Patterns		
Type	***Category***	***Pattern Name***
Design Patterns	Creational	Abstract Factory, Builder, Singleton, Lazy Initialization
	Structural	Front Controller, Composite, Adapter
	Behavioral	Template Method, Specification
Enterprise Application Patterns	Domain Logic	Domain Model, Service Layer
	Data Source Architectural	Data Mapper
	Object-Relational Behavioral	Unit of Work
	Object-Relational Metadata Mapping	Repository

	Web Presentation	Front Controller
	Distribution Patterns	Data Transfer Object (DTO)
	Base Patterns	Layer Supertype, Separated Interface
Enterprise Integration Patterns	Messaging Channels	Point-to-Point Channel Adapter
	Message Construction	Request-Reply
	Message Transformation	Content Enricher Content Filter Claim Check Canonical Data Model
	Composed Messaging	Synchronous (Web Services)
Design Principles		
SOLID Design Principles	**S**ingle Responsibility Principle (SRP) **O**pen/Closed **I**nterface Segregation **L**iskov Substitution (in conjunction with co- and contra-variance of generic types in .NET) **D**ependency Inversion (Data Access and Repositories are injected using MEF and Unity)	

IV. A Few Challenges and Solutions

This section captures a few interesting aspects that surfaced during the design and implementation of the API.

A. *Handling POST Requests for Large Numbers of Specialized Resource Types with Few URIs*

Simplified URI schemes have the benefit of providing a clean interface to consumers, without having to introduce a myriad of URIs, as would be the case of one URI per actual hardware device supported by the AIM system. The different representations of these resources are grouped by category, while specific details are handled using *custom* JSON *deserialization* behavior injected in the HTTP transport pipeline [3] [5]. Since all resources must specify the concrete entity type they represent (under the `ConcreteAssetTypeId` property), the custom deserialization framework can easily create instances of the *specialized* resource types based on this property, and pass them to the appropriate controller (one per URI/resource *category*) for handling.

The impact on performance is negligible given the use of a lookup dictionary mapping asset type ID to resource type, which is created only once (per app pool lifecycle) based on *metadata* defined on the model. Even if new specialized resource types are added, the lookup table will automatically be updated at the time the application pool is (re)started.

For example, a "360 iPatch Ultra High Density Fiber Shelf (2U)" and a "360 iPatch Modular Evolve Angled (24-Port)" [12] – two hardware devices that map to two different specialized types in the imVision API resource model, are both resources of type **`PatchPanel`**. Therefore, a POST request to create either of these will be sent to the same URI: **`http://[host:port/app/]PatchPanels`**.

This means that the same service components (controller and repository – and even stored procedure) will be able to handle either request but the API would also be aware of the distinction between these two different object instances, as created by the custom deserialization component.

B. *Adding Support for Composite Resources*

Hardware components are built as composite devices, containing child elements, which in turn contain sub-child entities. For example, the Quattro Panel contains six Copper Modules with each module containing exactly four Quattro Panel Ports. To realize these hardware-driven requirements and avoiding multiple POST requests, while preserving the integrity and correctness of the device representation, a rule-based composition representation model was used in conjunction with the Builder design pattern applied recursively down the object hierarchy. The composition rules for the Quattro Panel and its module sub-elements are shown in Figure 9 (using C#.NET). The strings represent optional name *prefixes* for the child elements.

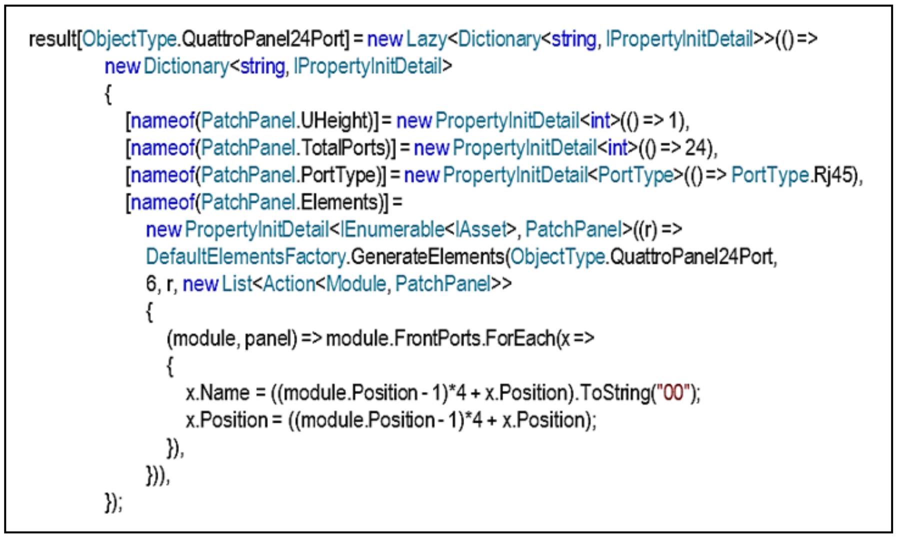
C. *A Functional and Rule-Based Approach for Default Initializations and Validations of Resources*

Given the considerable number of specialized resources to be supported by CommScope's imVision API and the even larger number of business rules regarding the initialization and validation of these entities, a functional approach was adopted. This rendered the validation engine into a rule-based system: there are *composition rules*, default *initialization rules*, and *validation rules* – which apply to both simple as well as complex properties that define a resource. Following the same example of Quattro Panel used earlier, an important requirement for creating such resources is the labeling of ports and their positions, which must be continuous across all six modules of the panel.

Figure 10 shows a snapshot of the rules defined for this type of asset. Figure 10 (a) shows the initialization rules whereas Figure 10 (b) shows a few of the validation rules. In both cases, the programming constructs like the ones shown make heavy use of lambda expressions as supported by the functional capabilities built into the C#.NET programming language [16], demonstrating the functional implementation approach adopted for the imVision AIM API.

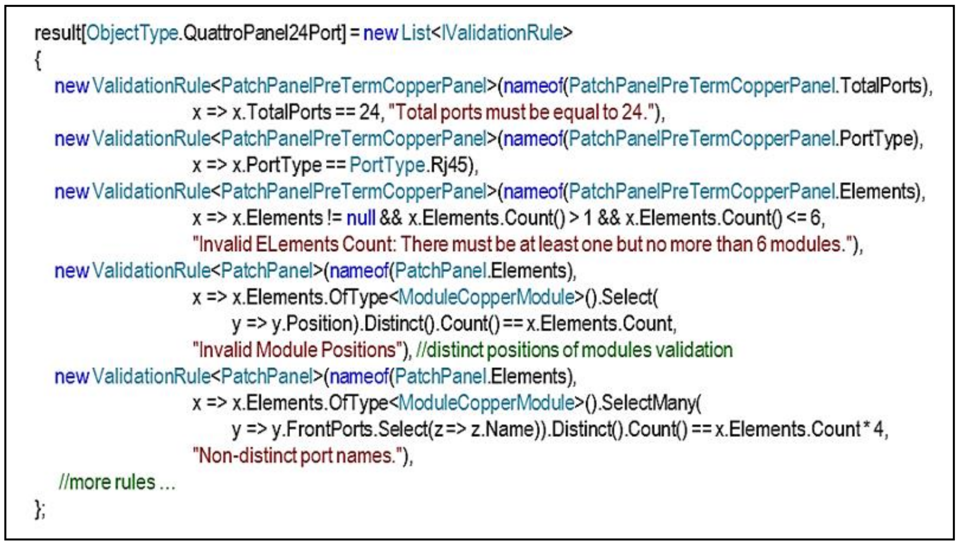
```
//…
{ ObjectType.QuattroPanel24Port,  new CompositionDetail<ModuleCopperModule,  int,  ModuleValidator>(ObjectType.CopperModule,  "Module", 6) },
//…
{ ObjectType.CopperModule,  new CompositionDetail<PortBasicPort,  int,  PortValidator>(ObjectType.QuattroPanelPort,  "Port", 4) },
//…
```

Figure 9. Composition Rules for Quattro Panel and Its Child Elements of Type Copper Module



```
result[ObjectType.QuattroPanel24Port] = new Lazy<Dictionary<string, IPropertyInitDetail>>(() =>
        new Dictionary<string, IPropertyInitDetail>
        {
          [nameof(PatchPanel.UHeight)] = new PropertyInitDetail<int>(() => 1),
          [nameof(PatchPanel.TotalPorts)] = new PropertyInitDetail<int>(() => 24),
          [nameof(PatchPanel.PortType)] = new PropertyInitDetail<PortType>(() => PortType.Rj45),
          [nameof(PatchPanel.Elements)] =
            new PropertyInitDetail<IEnumerable<IAsset>, PatchPanel>((r) =>
            DefaultElementsFactory.GenerateElements(ObjectType.QuattroPanel24Port,
            6, r, new List<Action<Module, PatchPanel>>
            {
              (module, panel) => module.FrontPorts.ForEach(x =>
              {
                x.Name = ((module.Position - 1)*4 + x.Position).ToString("00");
                x.Position = ((module.Position - 1)*4 + x.Position);
              }),
            })),
        });
```

Figure 10. (a) Default Initialization Rules Sample



```
result[ObjectType.QuattroPanel24Port] = new List<IValidationRule>
{
  new ValidationRule<PatchPanelPreTermCopperPanel>(nameof(PatchPanelPreTermCopperPanel.TotalPorts),
                    x => x.TotalPorts == 24, "Total ports must be equal to 24."),
  new ValidationRule<PatchPanelPreTermCopperPanel>(nameof(PatchPanelPreTermCopperPanel.PortType),
                    x => x.PortType == PortType.Rj45),
  new ValidationRule<PatchPanelPreTermCopperPanel>(nameof(PatchPanelPreTermCopperPanel.Elements),
                    x => x.Elements != null && x.Elements.Count() > 1 && x.Elements.Count() <= 6,
                    "Invalid ELements Count: There must be at least one but no more than 6 modules."),
  new ValidationRule<PatchPanel>(nameof(PatchPanel.Elements),
                    x => x.Elements.OfType<ModuleCopperModule>().Select(
                        y => y.Position).Distinct().Count() == x.Elements.Count,
                    "Invalid Module Positions"), //distinct positions of modules validation
  new ValidationRule<PatchPanel>(nameof(PatchPanel.Elements),
                    x => x.Elements.OfType<ModuleCopperModule>().SelectMany(
                        y => y.FrontPorts.Select(z => z.Name)).Distinct().Count() == x.Elements.Count * 4,
                    "Non-distinct port names."),
  //more rules ...
};
```

Figure 10. (b) Validation Rules Sample

Among some of the reasons worth mentioning for embracing the functional model are:

- ✓ a more robust, concise, reusable, and testable code
- ✓ minimizing side effects from object state management and concurrency.

Explicit goal specification – central to the functional programming paradigm – confers *clarity* and *brevity* to the rule definitions, as seen in the code samples provided here.

V. Integration with Other AIM Systems

As stated in the introduction, one of the main goals of the API is to allow easy integration between AIM systems. This section presents a theoretical approach to such an integration.

A. *Quareo Middleware API*

CommScope's Quareo physical layer management solution is a real-time physical connectivity provisioning system with a dual hardware and software implementation [17]. Using an eventing mechanism, Quareo provides immediate feedback on all network connection elements, while enabling technicians to efficiently and accurately respond to address a variety of infrastructure connectivity concerns and responsibilities.

Originally, Quareo was developed under one of TE Connectivity's units – which was acquired by CommScope in 2015. Now, two similar yet different systems provide comparable services to CommScope's clients and, not surprisingly, the need to unify the two systems' functionality of provisioning managed connectivity data has become a recognized necessity and focus for the company.

The Quareo Middleware API exposes networking infrastructure elements via a RESTful API but the focus is exclusively on telecom assets. It is also using a more generic approach to modeling these elements than does imVision.

The next sub-section briefly describes the main resource models as designed and implemented for the Quareo system.

B. Quareo Resource Models

A more generic representation of telecom assets makes the API more flexible and extensible. However, this puts a burden on the model itself to allow for this generalization – requiring potentially a more complex *mapping* of concrete assets to generic elements which may or may not be able to describe all the attributes of the assets in a straightforward and strongly-typed fashion. Additional complexities may arise on the consumer end; integrators must have sufficient detail as of how to restore specialized hardware information from the generalized representation and how new hardware elements will be represented by the system.

From a high-level perspective, some of the main entities of the resource model employed by the Quareo Middleware API are shown in Figure 11. Since the hardware assets that the middleware provisions feature the Connection Point Identification (CPID) technology [18], all assets (including the most granular of elements, i.e., **Port**) inherit from the base class **CpidComponent**, which encapsulates a large array of hardware-specific attributes – modeled as simple or complex types, such as color, connector type, copper/fiber cable category/rating/polarity, manufacturer Id, hardware revision, insertion count, catalog, etc.

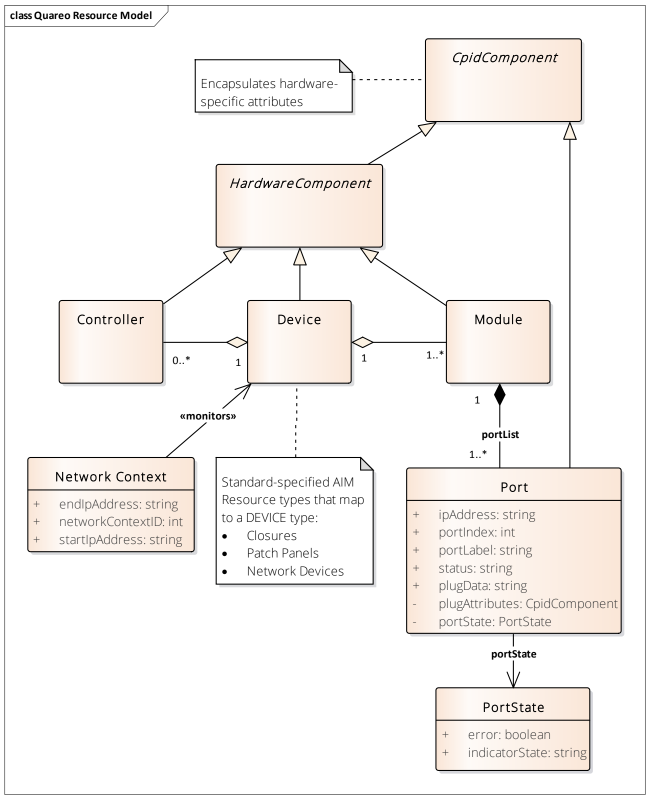


Figure 11. Simplified View of Quareo's Telecom Assets Resource Model

C. AIM Software Systems Integration Scenarios

1) One-way Integration

Assuming one of the systems as the system-of-record, or primary infrastructure provisioning system, a one-way integration solution could be devised such that telecom assets provisioned by the secondary system can be retrieved via RESTful GET API requests by the designated primary system. Consequently, the infrastructure elements managed by the secondary system become visible to the primary system. Given that imVision currently provisions more than just the telecom assets, it would be an obvious choice for being considered as the primary system in this proposed integration solution. This would include having its resource model become the canonical model for all data exchange – as described later.

Pulling the data managed exclusively by Quareo into imVision can either be (a) a one-time operation - which would then require managing connectivity via imVision only, or (b) a periodic process which would allow the Quareo system to continue managing telecom assets while imVision would only be allowed to report on these assets.

Figure 12 shows the general integration scenario and the data flow between these two systems.

2) Bidirectional Integration

If a unified collection of networking resources is to be managed by more than one software system, assuming that each system enables some highly-specialized set of features that would be prohibitively expensive to migrate to the other system, then data – and (to a lesser extent) functional – integration concerns would be applicable at both ends. If only two systems are considered, then a direct point-to-point integration mechanism via the already exposed integration APIs is possible and recommended.

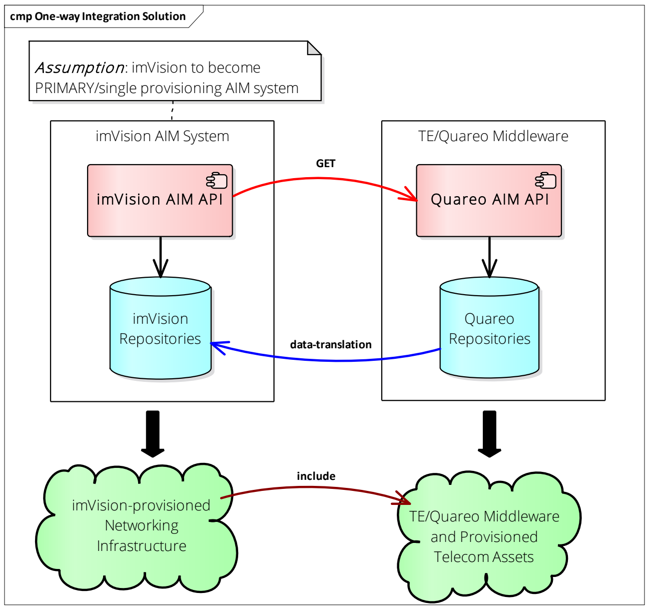

cmp One-way Integration Solution
Assumption: imVision to become PRIMARY/single provisioning AIM system
imVision AIM System
TE/Quareo Middleware
GET
imVision AIM API
Quareo AIM API
imVision Repositories
Quareo Repositories
data-translation
include
imVision-provisioned Networking Infrastructure
TE/Quareo Middleware and Provisioned Telecom Assets

Figure 12. A Straightforward One-Way Integration Scenario

However, adding even one more system to the mix, an integration infrastructure would be required in order to reduce complexity and isolate the integration responsibilities and models. A similar problem, where multiple business domains required both data and functional integration across the enterprise, has been presented and discussed in [19]. For the CommScope integration scenario, a comparable framework based on messaging and eventing communication mechanisms would also be appropriate, especially given the real-time nature of Quareo's solution as opposed to the offline update features of imVision.

D. Data Integration Approach and Challenges

The following discussion assumes the adoption of the first integration option, where imVision system will be provisioning the entire infrastructure using a persistent representation of all the telecom assets, as well as the premise, organizational, and container elements.

1) Data Model Refactoring

In order to support specialized hardware attributes featured by the CPID technology specific to Quareo [18], the data layer currently used to model assets managed by the imVision system must be refactored and enhanced accordingly. For example, Table IV shows the additional Quareo attributes that were extended to the imVision data model for Port and Cable assets. They can also be seen in the Entity Relationship Diagram (ERD) in Figure 13, which defines these attributes under the two corresponding tables.

As part of a comprehensive architectural effort, a set of data model updates were designed and recommended. Figure 13 highlights a *sample* output of such data model refactoring – more precisely, the models corresponding to the main asset connectivity elements (ports and cables).

TABLE IV. QUAREO-SPECIFIC CABLE AND PORT ATTRIBUTES

Entity	***Required Attributes***
Port Detail	Color Insertion Count Indicator State Is Managed
Cable Detail	Color Hardware Revision Serial Number Country of Manufacture Date of Manufacture Manufacturer Part Id Material Tracking Number

While the data models capture all relevant attributes of the hardware assets that they represent, other essential data layer design requirements and concerns were addressed as part of this effort. To provide a few pointers as to what this effort entailed, the list below highlights some of the data layer refactoring tasks that were undertaken:

- ✓ Appropriate entity relationship modeling (realized by adding the missing foreign keys)
- ✓ Data integrity and referential integrity (also done via foreign keys and unique constraints)
- ✓ Careful data type selection/updates
- ✓ Lookup data specification for modeling essential domain attributes
- ✓ Normalization and vertical partitioning of tables to reduce data redundancy among entities that share similar attributes
- ✓ Refactoring of functions and procedures; moving stored procedure implementations to table-valued functions where no data mutation was involved
- ✓ Proper schema partitioning, associations and object renaming, while creating synonyms for these objects in order to reduce the impact on the application layer
- ✓ Data cleansing required to remove *existing* invalid data or data that would not conform to the added constraints.

2) Representation and Identification of Telecom Assets

Both AIM systems use integer-valued surrogate keys to identify the provisioned entities but – in order to migrate data from one system to the other, or to continuously exchange data between the two – it is imperative to identify attributes that uniquely identify concrete hardware asset types. Fortunately, the ISO/IEC standards document has accounted for such *natural* keys based on serial numbers, manufacturer details, and component identifiers. Since these attributes have been included into the various models (data, domain, resource), an integration solution would no longer require a resolution framework where cross-domain asset identifiers would be stored and looked up. Serial numbers and perhaps manufacturing data will represent the business domain identifiers that integration adapters on both sides of the integration boundary would use when adding, updating and deleting asset data from the corresponding repositories.

However, model translators/adapters are required on both sides since the asset representations are quite different, but not irreconcilable – given the semantics imposed by the Standards specification and the extent to which they are implemented by each system participating in the integration.

To alleviate the constant need for updating these adapter components whenever new hardware must be handled by the AIM systems, extensible models and intelligent mapping frameworks could be created; ideally, these would be encapsulated under distributable and reusable model brokers that are capable of bi-directional data translation and consistent asset type resolution.

Schematically, this brokered adaptive integration layer would look similar to the one depicted in Figure 14. The AIM systems will not depend directly on each other's asset representation but rather delegate the translation task to the adapter component.

Given that only two systems are involved in the message exchange, there is no need for designing a common model to normalize the two representations of assets. However, to facilitate future integration needs, it would be beneficial to designate the more comprehensive model as the *canonical representation* of telecom assets, expose it to all integrating systems, and use it as the *ubiquitous integration language* across the enterprise, analogous to the approach described in [19], following a pattern quite similar to the *Normalizer* messaging EIP (Enterprise Integration Pattern) [14].

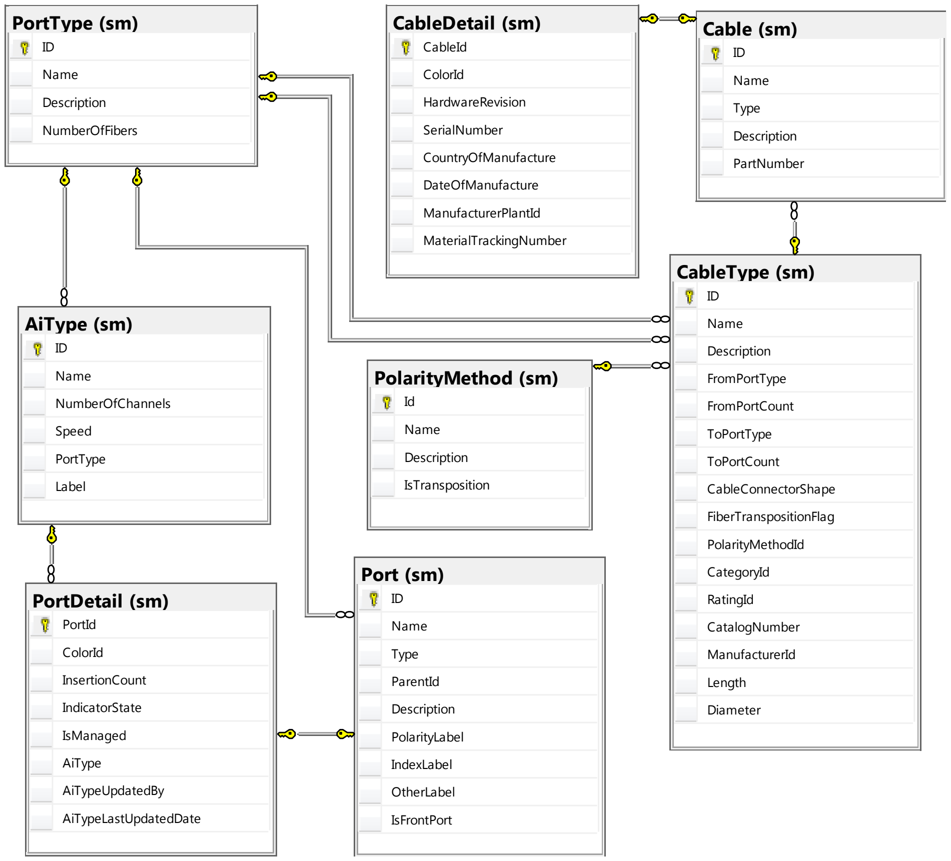


Figure 13. The Entity Relationship Diagram (ERD) Including the Refactored Connectivity Elements

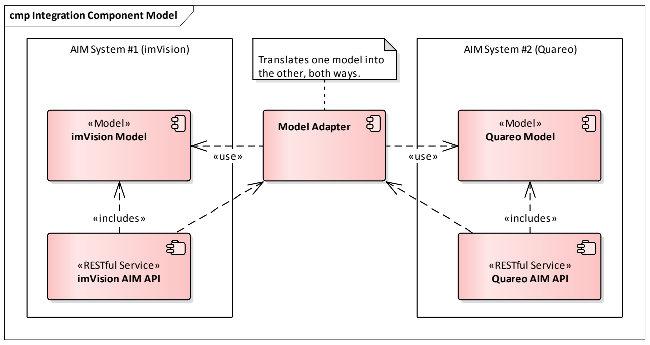


Figure 14. Enabling Data Exchange via a Model Adapter

VI. Conclusion

Modeling large varieties of telecommunication assets can be a challenging task, even more so if other applications intend to integrate with one or more systems that automate the management of such complex telecommunication enterprise infrastructure and their physical connectivity.

The benefits entailed by the model standardization of the entities managed by such systems are significant and can be summarized as follows:

- ✓ Standardized models facilitate a common understanding of the AIM systems in general and of the elements that such systems expose and provision;
- ✓ The common model is divorced from any proprietary representation of telecommunication assets while still allowing the inclusion of vendor-specific details;
- ✓ The ISO/IEC specifications define a true domain model of the physical layer connectivity;
- ✓ The model is technology-agnostic;
- ✓ By omitting unnecessary detail, the model is highly flexible, allowing both present and future network hardware specification in a unified fashion;
- ✓ The ISO/IEC standardization enables and ensures a systematic, consistent, and unified modeling of AIM systems;
- ✓ Functional features of AIM systems, such as connectivity provisioning and asset management, can easily be described and modeled in terms of the structural elements introduced by the Standards document;
- ✓ Integrating with AIM systems is a considerably less complex undertaking, given the standardized model that systems can now use to communicate with each other.

This paper took further steps to elaborate on these models and the relationships between them via concrete design artifacts developed using UML (Unified Modeling Language). Inheritance, composition/aggregation, and generic typing were used in designing a hierarchical resource model shown to be extensible and fit for representing telecommunication assets, connectivity features and activities, premises, organizational elements, and system notifications – as they relate to any AIM-centric domain.

Although the primary focus of the 18598/DIS draft ISO/IEC Standards document is to address the representation of network connectivity assets, the motivation behind this specification is to facilitate custom integration solutions with AIM systems. Given the challenging nature of software systems integration in general, building AIM systems with the right quality attributes that support such integration is essential. Extensibility, scalability, rigorous and stable interface and model design, and performance through adequate technology adoption are important goals to consider. For this reason, the present paper also introduced the layered architecture adopted by CommScope's imVision API, targeting the management of telecommunications infrastructure.

Emphasis was placed on the Standards-recommended RESTful architectural style, while technology specifics were succinctly described to show how they helped align the system's design and functionality with the AIM standards requirements. Various design and implementation aspects were elaborated along with a selection of key benefits, such as dynamic resource composition, custom serialization to support consistent handling of similar resources, efficient POST request construction and network traffic, and a simple URI scheme despite large varieties of specialized resources.

Delving into a few technology-specific facets, a brief overview of a rule-based engine and supporting frameworks designed for resource initialization and validation was described. Interesting implementation details that highlight aspects of the functional programming paradigm employed by key components of CommScope's imVision API were also shared.

Considering the imVision and Quareo resource models, the AIM API architecture, and exposed features of the CommScope's networking infrastructure provisioning system, integration-related aspects were also addressed. Data integration concerns were considered for the imVision software system as they were tackled as part of the data layer refactoring effort prompted by non-functional requirements such as extensibility, robustness, and – last but not least – organizational data integration needs.

A straightforward integration candidate solution between CommScope's imVision AIM API and Quareo API RESTful services was presented – one based on model normalization and point-to-point messaging. Both one-way/one-time and two-way integration scenarios were discussed, concluding with a brief debate regarding the need for a canonical model to allow the AIM systems to efficiently communicate with each other.

VII. REFERENCES

[1] M. Iridon, "Automated Infrastructure Management Systems. A Resource Model and RESTful Service Design Proposal to Support and Augment the Specifications of the ISO/IEC 18598/DIS Draft," FASSI 2016 : The Second International Conference on Fundamentals and Advances in Software Systems Integration, ISBN: 978-1-61208-497-8, pp. 8-17, Nice, France, July, 2016.

[2] Automated Infrastructure Management(AIM) Systems– Requirements, Data Exchange and Applications, 18598/DIS draft @ ISO/IEC.

[3] G. Block et. al., "Designing Evolvable Web APIs with ASP.NET," ISBN-13: 978-1449337711.

[4] J. Kurtz and B. Wortman, "ASP.NET Web API 2: Building a REST Service from Start to Finish," 2nd Edition, 2014, ISBN-13: 978-1484201107.

[5] J. Webber, "REST in Practice: Hypermedia and Systems Architecture," 1st Edition, 2010, ISBN-13: 978-0596805821.

[6] E. Evans, "Domain-Driven Design: Tackling Complexity in the Heart of Software," 1st Edition, Prentice Hall, 2003, ISBN-13: 978-0321125217.

[7] Microsoft, "Microsoft Application Architecture Guide (Patterns and Practices)," Second Edition, Microsoft. ISBN-13: 978-0735627109. [Online] Available from: https://msdn.microsoft.com/en-us/library/ff650706.aspx [retrieved: March 2016].

[8] M. Fowler, "Patterns of Enterprise Application Architecture," Addison-Wesley Professional, 2002.

[9] T. Erl, "Service-Oriented Architecture (SOA): Concepts, Technology, and Design," Prentice Hall, 2005, ISBN-13: 978-0131858589.

[10] R. Daigneau, "Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services," Addison-Wesley, 1st Edition, 2011, ISBN-13: 078-5342544206.

[11] M. Fowler, "The Richardson Maturity Model". [Online]. Available from http://martinfowler.com/articles/richardsonMaturityModel.html [retrieved: March 2016].

[12] CommScope Enterprise Product Catalog. [Online] Available from: http://www.commscope.com/Product-Catalog/Enterprise/ [retrieved March 2016].

[13] G. M. Hall, "Adaptive Code via C#: Agile coding with design patterns and SOLID principles (Developer Reference)," Microsoft Press, 1st Edition, 2014, ISBN-13: 978-0735683204.

[14] G. Hohpe and B. Woolf, "Enterprise Integration Patterns; Designing, Building, and Deploying Messaging Solutions," Addison-Wesley, 2012, ISBN-13: 978-0321200686.

[15] M. Seemann, "Dependency Injection in .NET," Manning Publications, 1st Edition, 2011, ISBN-13: 978-1935182504.

[16] T. Petricek and J. Skeet, "Real-World Functional Programming: With Examples in F# and C#," Manning Publications; 1st edition, 2010, ISBN-13: 978-1933988924.

[17] CommScope Quareo Physical Layer Management System. [Online] Available from: http://www.commscope.com/Docs/Quareo-Physical-Layer-Management-System-BR-319828-AE.pdf [retrieved February 2017].

[18] CommScope NG4access ODF Platform [Online] Available from: http://www.commscope.com/Docs/NG4access_ODF_Platform_Quareo_CO-319580-EN.pdf [retrieved February 2017]

[19] M. Iridon, "Enterprise Integration Modeling – A Practical Enterprise Integration Solution Featuring an Incremental Approach via Prototyping," International Journal on Advances in Software, vol. 9 no. 1&2, 2016, pp. 116-127.