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

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 that enable network infrastructure to operate locally and across vast geographical areas. The representation of infrastructure elements managed by such systems has never been normalized before, making integration – a challenging undertaking on its own – an even more difficult task, requiring specialized knowledge about the systems and the infrastructure data they provision. Such details are most relevant given the complexity and variety of telecommunication infrastructure systems and the widespread need for external or custom applications to gain access to the data and features built in to these AIM systems. This year however, the international standards organization is scheduled to release new standard ISO/IEC 18598 that will provide standardization and sensible guidelines for exposing data and features of AIM systems and thus to facilitate the integration with custom clients for these systems. CommScope, an active contributor in defining these standards, has implemented to a large extent these specifications for their imVision system and in doing so, decided to capture some relevant details that would bring more clarity, add context, and provide further guidelines to the information described by the standards document. In order to achieve these goals and in an attempt to lead the way towards a robust AIM system design that aligns with these standards, this paper elaborates on the recommended models. It also intends to share architectural and technology-specific considerations, challenges, and solutions adopted for the CommScope's imVision standards-based API, 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; ISO/IEC 18598.

I. INTRODUCTION

ISO/IEC have recently put forth a set of requirements and guidelines for modeling and provisioning Automated Infrastructure Management (AIM) systems [1] that will help consolidate how such systems represent the assets and entities they provision, as well as enable custom integration solutions with these systems. 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.

As with every software system and more so with enterprise-level applications, domain modeling is of crucial importance as it helps define, refine, and understand the business domain, facilitating the translation of requirements into a suitable design [5]. However, special-purpose models can and should be designed for various layers of a system's architecture [11]. When a system exposes integration points to outside agents or clients, it is imperative to define clean boundaries between the system's domain and the integration models [6] [4]. Stability of integration models is just as important as versioning for extensible systems, while allowing the domain models, structural or behavioral, to evolve independently of all other models that the system relies on [2] [3].

The first half of this paper (Section II) will present the relevant resource models from the perspective of a RESTful services design [2] [10] [13], with focus on the underpinning structures and 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 a large number of URIs for accessing these resources. Section III discusses system architecture, patterns and design-specific details. Section IV presents some of the challenges encountered during the realization of the system design, solutions employed, and finally joining all the discussion points to a conclusion in Section V.

II. AIM SYSTEM DOMAIN ANALYSIS AND RESOURCE MODELING

The resource model presented in this paper employs various design and implementation paradigms. However, the only types exposed by the system, i.e., all concrete resource types, can be viewed and modeled as 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 include any behavior whatsoever. Such features are specific to the physical entities being modeled and are highly customized for a given system. The model proposed here serves the purpose of defining a common understanding of the data that can be exchanged with an AIM system while any specific

 ISBN: 978-1-61208-497-8

behavior around these data elements is left to the implementation details of the particular AIM system itself.

As opposed to stateful services design principles (such as SOAP and XM -RPC-based web services) - where functional features and processes take center stage while data contracts are just means to help model those processes [4] [11], 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 [7] [13]. 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.) [10] [13]. Nonetheless, in both cases, a sound design principle (as with any software design activity in general) is to remain technology-agnostic [5] [6] [11].

A. *Resource Categories Overview and Classification*

The entities proposed in the Standards document [1] are categorized by the sub-domain that they are describing as well as their composability features. At the high-granularity end of the spectrum we will find entities that deal with the location of networking centers (sites, cities, buildings, rooms, etc.) while at the other end of the spectrum we have the smallest assets that the system manages (modules and ports, outlets and cables). 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 Standards document proposes the following categories of resources to be provisioned by an AIM System, as shown in Table 1.

TABLE I. RESOURCE CATEGORIES AND EXAMPLES OF 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

Some elements listed above may not be relevant to all AIM systems. The Standards document intends to capture and categorize all elements that could be modeled by such a system. It also suggests 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 and what their purpose is. Otherwise stated, it defines at high-level the ubiquitous integration language by providing a clear description and classification of the main elements of an AIM system. This paper takes these recommendations, materializes them into actual design artifacts, and proposes a general-purpose layered architecture for the RESTful AIM API system.

B. *Common Abstraction Models*

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 identifiers (i.e., Globally Unique Identifier, integer, string, etc.) the `ResourceBase` class is modeled as a generic type, with the resource and parent identifier values of generic `TId`.

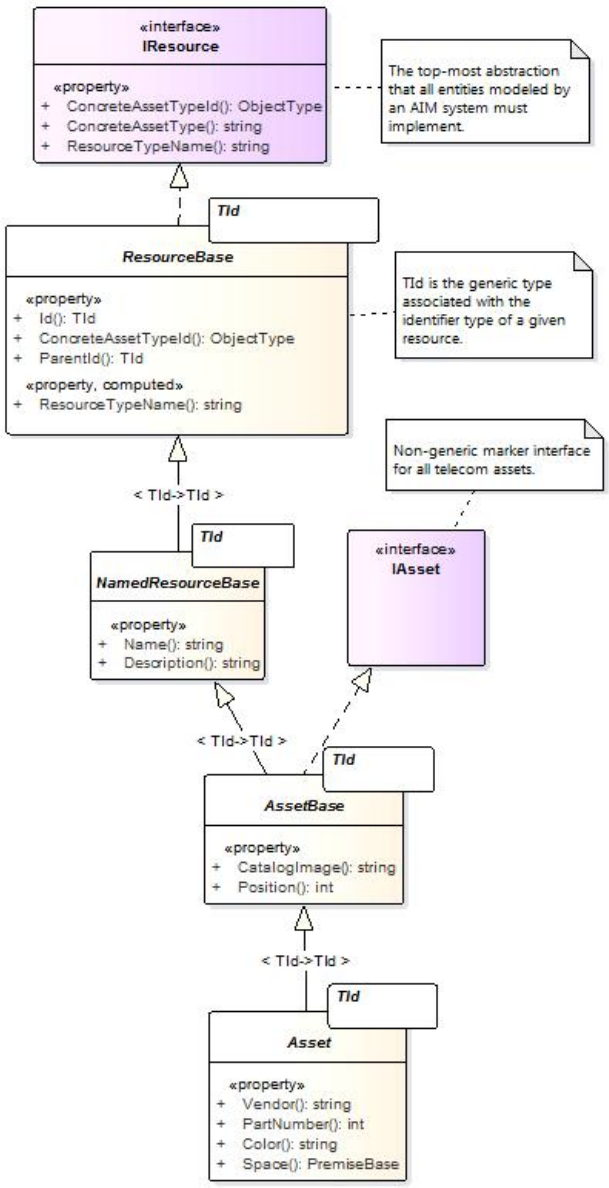

Figure 1. Resource Base Models

Of particular interest are telecommunication assets – the core entities in AIM systems – a class of resource types, which all realize the `IAsset` interface, an abstraction used as a marker on the type. These entities will be presented in the next sub-section.

C. Resource Model Design

1) Premise Elements

Company'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 the 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. 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 object 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 enable circuits to be specified: cables, connectors, and cords. They play a 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

Some AIM systems may desire to 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 that comprise one or more work order tasks. Such tasks are then assigned to technicians, which report to a manager, which in turn belongs to a department, and so on. The model for these elements is not included here as it is straightforward but is available upon request.

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 synchronization of the hardware state with the logical representation of the hardware components.

This synchronization is facilitated by the concept of events and alarms that are first generated by controllers (alarms) and then sent for processing by the management

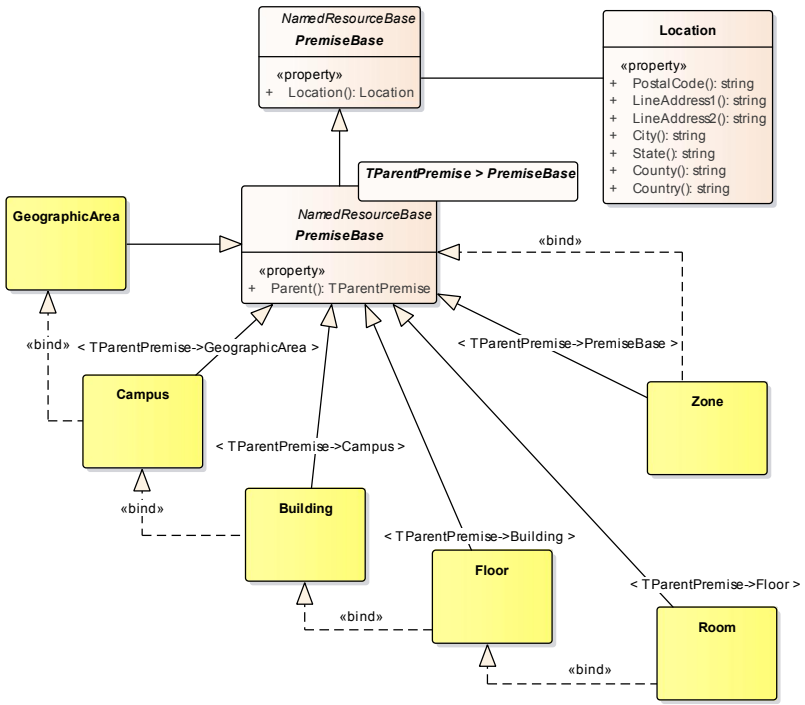

Figure 2. Premise Resource Models

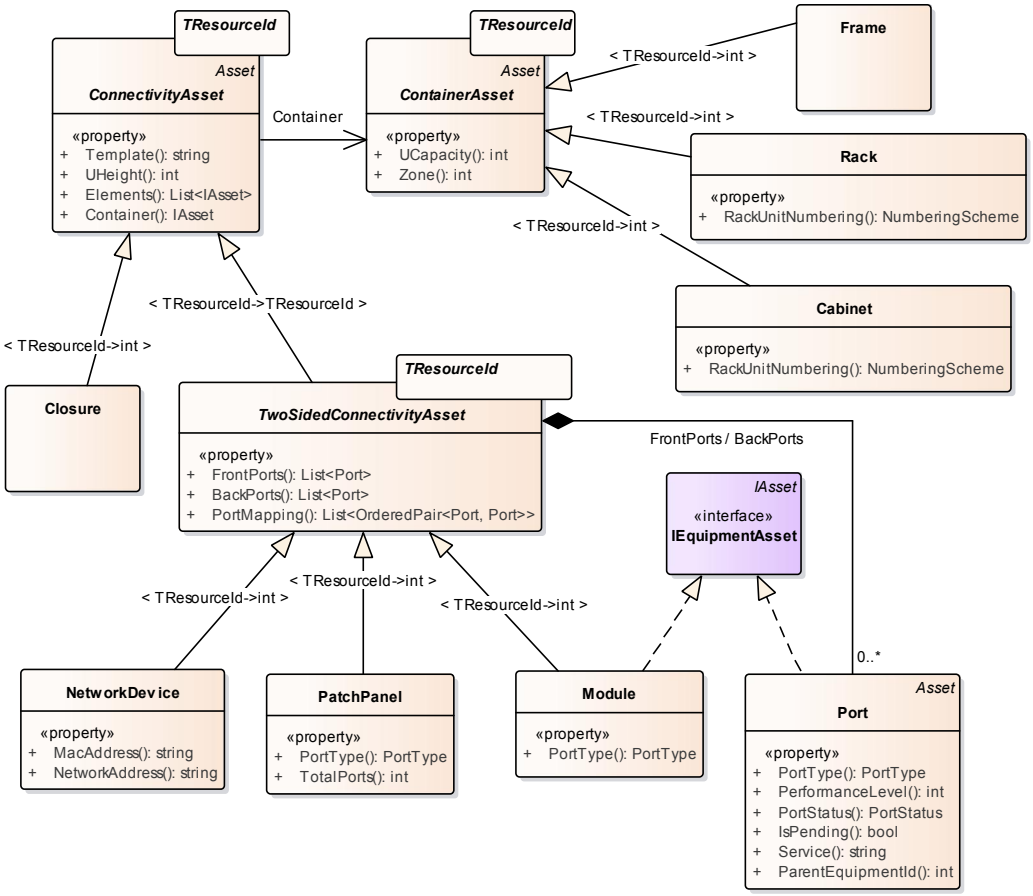

Figure 3. Telecommunication Assets Resource Models

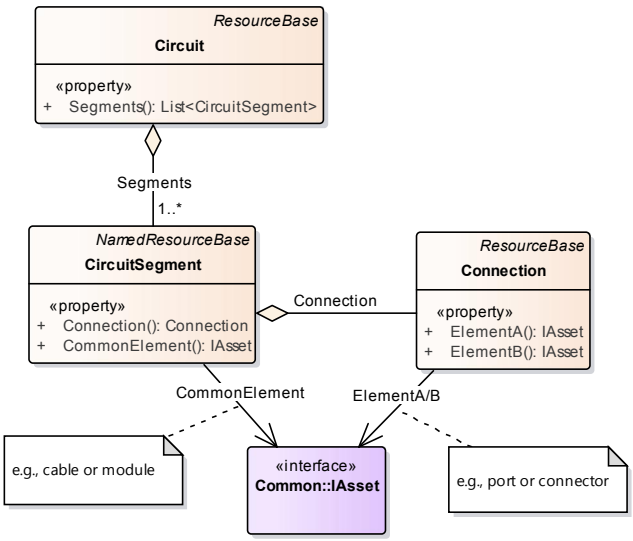

Figure 4. Connectivity Models

Figure 5. A JSON Representation of a Rack Resource

software (events). These notification resource types are supported by the AIM Standards and are modeled as shown in Figure 6. The figure 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.

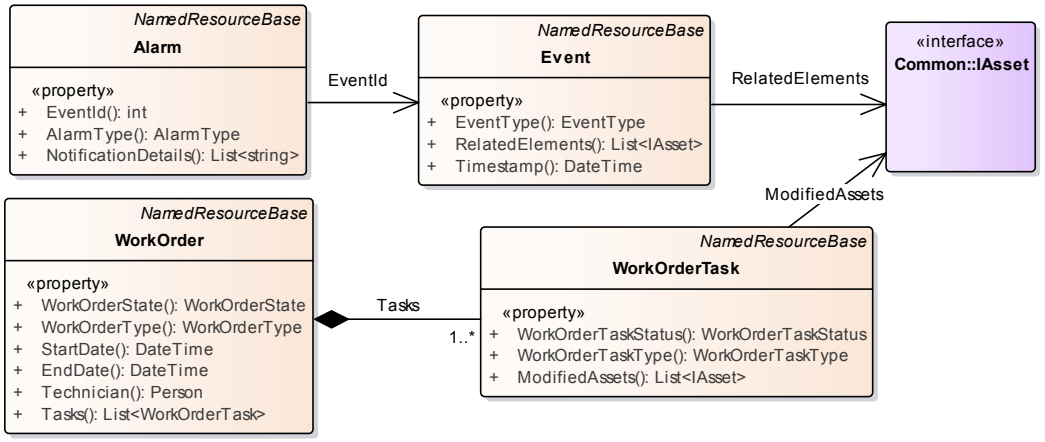

Figure 6. Notification and Activity Models

D. *Modeling Large Varieties of Hardware Devices*

The telecom asset model presented in Figure 3 depicted 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 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 is how to support such a large variety of hardware devices without having to expose too many different service endpoints for each of these specialized types.

According to the Richardson Maturity Model for REST APIs [7], 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, we have to be very clear about which concrete entity or device type we want to create, and for this, we must provide the device-specific data. Since these features are not inherent to all objects that belong to that category, specialized models must be created – e.g., as derived types from the category models that encapsulate all relevant device-specific features.

For example, one of CommScope 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 [15] – 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 differentiation between the various hardware components that map to the same specialized type can be managed, for example, via metadata associated with that data type (e.g., the `AllowedObjectTypeAttribute` in Figure 7).

This approach saves us from having to define one data type per physical device type and furthermore, allows accessing a variety of devices that fall under the same category, using the same URI – as described in the next sub-section.

E. *Benefits of the Proposed Model*

The models proposed in this paper are closely following the categories and entities outlined by the ISO/IEC standards. However, given the structural models presented here and taking advantage of available technology-specific constructs and frameworks, select design features exist that confer certain advantages to these models, to their usage, and the integration capabilities for the services that expose them, with direct impact on performance, maintainability, testability, and extensibility.

 ISBN: 978-1-61208-497-8

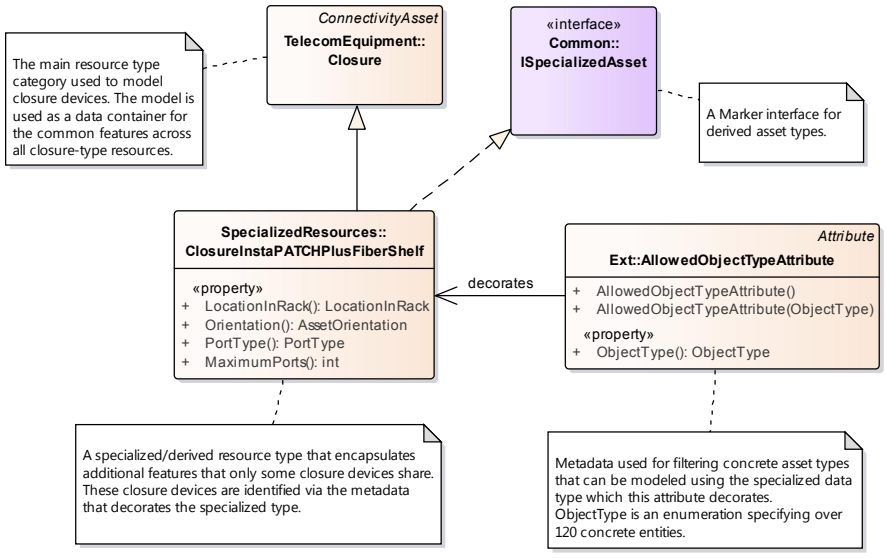

Figure 7. A Sample Specialized Closure with Additional Properties

- ✓ *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 and consistency 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 [8]), 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 to be created, or they can be omitted altogether, while – in the case of the imVision API - the Validation and Composition frameworks would take care of filling in the missing sub-resources based on predefined composition and default initialization rules.

  Table 2 captures just a few but noteworthy metrics regarding the request counts and sizes for creating a complete resource of a specialized `PatchPanel` type.
- ✓ *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 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

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

the process depends on properties that the main resource may expose (e.g. `TotalPorts`) – which client applications will have to specify if the property is marked as [`Required`], but the composing port sub-elements may be omitted from the request body, as they will be automatically created and added.

- ✓ *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. The design for adding and updating resources follows the Open/Closed principle [8], so that new types, properties, and rules will be added or extended but existing ones will not change, ensuring contract stability.

III. A PROPOSED LAYERED ARCHITECTURE FOR AIM API INTEGRATION SERVICES

A. Adding Integration Capabilities to an AIM System

As per the Standards document guidelines, 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 [6] [11], 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 [8], this interaction takes place via interfaces, allowing for a loosely coupled system [9], easy to maintain, test, and extend. This will also enable the use of dependency injection technologies such as Unity, MEF, AutoFac, etc., to create a modular, testable, and coherent design [12].

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 to the existing system. The layered design of this new service component 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 very limited extent - a few components from the existing imVision system that encapsulate reusable logic.

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 component upon which the POST functionality relies. Along with the resource composition and validation engine, they constitute in fact a highly specialized rule-based system that makes extensive use of several design and enterprise integration patterns that will be discussed next.

B. Patterns and Design Principles

The various patterns and principles [6] [8] [9] employed throughout the design and implementation of the imVision API system are summarized in Table 3. The automation capabilities baked into the imVision API mentioned earlier, that support creating composite resources, are a direct realization of the Content Enricher integration pattern used in conjunction 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.

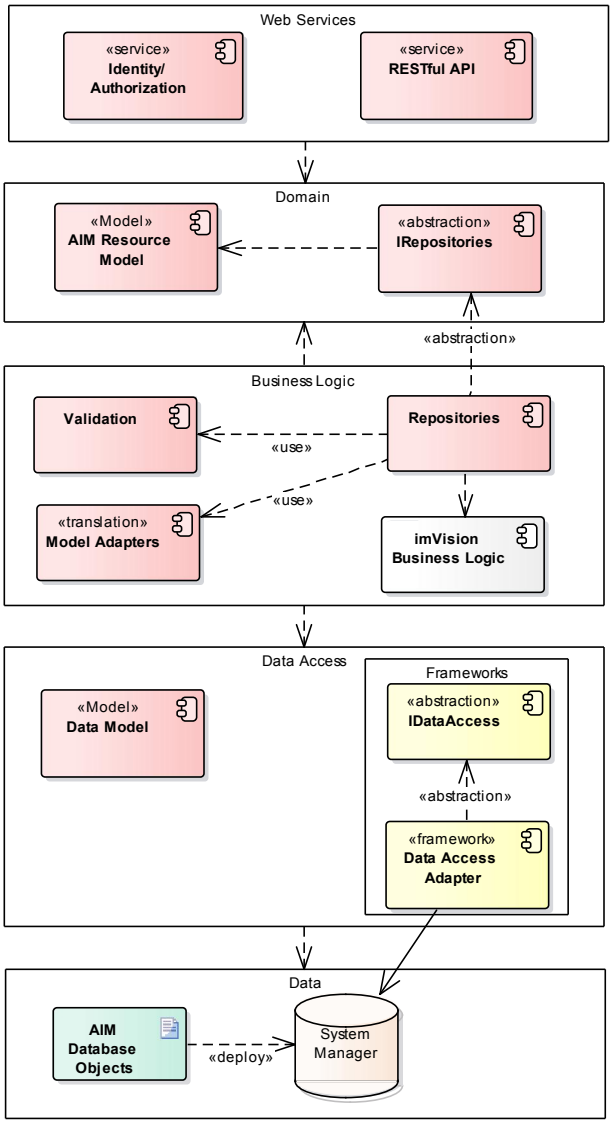

Figure 8. The Layered Architecture of the imVision AIM API

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

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, one per actual hardware device supported by the AIM system.

As shown in Section II, the different representation of these resources are grouped by category, while specific details are handled using *custom* JSON *deserialization* behavior injected in the HTTP transport pipeline [2] [13]. 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).

The impact on performance is negligible given the use of a lookup dictionary of 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 instantiated (restarted), ensuring the inherent extensibility of the custom deserialization framework.

This way, whether a user would like to create a "360 iPatch Ultra High Density Fiber Shelf (2U)" or a "360 iPatch Modular Evolve Angled (24-Port)" [15], even though these two hardware devices map to two different specialized types in the imVision API resource model, they 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) 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 of the device representation, a rule-based composition representation model was used in conjunction with the Builder design pattern applied recursively.

The composition rules for the Quattro Panel and its module sub-elements are shown in Figure 9 (The strings represent optional name prefixes for the child elements.).

```
//…
{ 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

 ISBN: 978-1-61208-497-8

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

Given the large 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* (above), default *initialization rules*, and *validation rules* (below) – which refer 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 that the panel contains.

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 some 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 [14], demonstrating the functional implementation approach adopted for the imVision API.

Among some of the reasons worth mentioning for taking the functional route are a more robust, concise, reusable, and testable code, and 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, both evident in the code samples provided hereby.

```
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

 ISBN: 978-1-61208-497-8

V. 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. The benefits entailed by the standardization of modeling entities managed by such systems are significant, as they facilitate a common understanding of the AIM system in general and the elements it exposes, their functional features, and their internal makeup. ISO/IEC proposed such standardization for a more systematic and unified modeling of AIM systems. This paper took further steps to present detailed models and the relationships between them using design artifacts modeled via UML (Unified Modeling Language). Using inheritance, composition/aggregation, and generic typing, a hierarchical resource model was designed and shown to be extensible and fit for representing telecom assets, connectivity, premises, organizational elements, and system notifications – as they relate to any AIM-centric domain.

Although the focus of the 18598/DIS draft ISO/IEC Standards document is to unify 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 integration in general, building AIM systems with integration in mind 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.

Finally, a very brief overview of a rule-based engine for resource initialization and validation was described, along with some implementation details that highlight aspects of the functional programming paradigm employed by key components of CommScope's imVision API.

VI. REFERENCES

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

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

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

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

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

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

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

[8] 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.

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

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

[11] 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].

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

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

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

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

 ISBN: 978-1-61208-497-8