|  |  |  |
| --- | --- | --- |
|  | Document Number | EDCS-XXXXXX |
| Based on Template | N/A |
| Created By | Enrico Schiattarella  Jay Weinstein |

Insieme Model-Driven Framework

A functional description of the Insieme Model-Driven Framework,  
covering fundamental concepts, key design principles and main features

Reviewers

|  |  |
| --- | --- |
| Department | Name/Title |
| Software | Mike Dvorkin, Jay Weinstein, Shrey Ajmera |
|  |  |
|  |  |
|  |  |

Modification History

|  |  |  |  |
| --- | --- | --- | --- |
| Revision | Date | Originator | Comments |
| 1 | 07/18/2012 | Enrico Schiattarella | First draft |
| 2 | 07/25/2012 | Jay Weinstein | Added class relation, catalog, api details |
| 3 | 09/20/2012 | Jay Weinstein | Updated details on explicit and named relations |

Table of Contents

[Insieme Model-Driven Framework 1](#_Toc335926899)

[Reviewers 1](#_Toc335926900)

[Modification History 1](#_Toc335926901)

[1 Introduction 3](#_Toc335926902)

[1.1 Model-Driven Systems 3](#_Toc335926903)

[1.2 Management Principles 4](#_Toc335926904)

[1.3 Framework components 4](#_Toc335926905)

[1.4 Languages, libraries and technologies 5](#_Toc335926906)

[2 Fundamental Concepts 7](#_Toc335926907)

[2.1 Object Model 7](#_Toc335926908)

[2.1.1 Managed Objects 7](#_Toc335926909)

[2.1.2 Management Information Tree 7](#_Toc335926910)

[2.1.3 Object Relationships 7](#_Toc335926911)

[2.1.4 Property attributes 10](#_Toc335926912)

[2.1.5 Packages 10](#_Toc335926913)

[2.1.6 Behaviors 10](#_Toc335926914)

[2.1.7 Metadata 10](#_Toc335926915)

[2.2 Building blocks for management systems 11](#_Toc335926916)

[2.2.1 Organizations 11](#_Toc335926917)

[2.2.2 Policies 11](#_Toc335926918)

[2.2.3 Pools 11](#_Toc335926919)

[2.2.4 Templates 12](#_Toc335926920)

[2.2.5 Faults 12](#_Toc335926921)

[2.2.6 Capability Catalog 12](#_Toc335926922)

[3 Application Programming Interface 13](#_Toc335926923)

[3.1 API Methods 14](#_Toc335926924)

[3.2 Stimuli and Transactions 15](#_Toc335926925)

[3.2.1 ACID properties 15](#_Toc335926926)

[3.2.2 Other API properties: idempotency, forgiveness 16](#_Toc335926927)

[3.3 Generic Object Notifications 17](#_Toc335926928)

[4 References 18](#_Toc335926929)

# Introduction

The Insieme Model-Driven Framework is an object-oriented software framework aimed at the development of management systems. In its previous incarnations it has been used to develop management applications for servers [SAM-FS], routers and other networking and storage equipment.

The framework constitutes the foundation for Insieme Fabric Controller (IFC) software and is used both in the controller and in policy-elements. Examples of framework-based components include Policy Manager, End-point DB, Fault Manager and Observer [IFC-FS].

The purpose of this document is to introduce the basic concepts and the key design principles of the framework. It surveys the main features and explains how they characterize the systems built on top. It does not get into any detail about how functionality is implemented or is supposed to be used by developers. For such information, the best reference currently is [UCSM-DEV].

Intended audience is members of the Insieme software team and reviewers of the IFC specs.

## Model-Driven Systems

In model-driven architectures, software maintains a complete, explicit representation of the administrative and operational state of the system (the model) and performs actions only as side-effects of mutations of model entities. Domain logic is expressed as a set of call-back functions to be invoked in response to model changes as opposed to procedures implementing specific functionality. Model changes can be caused by a variety of events, such as user-initiated configuration changes, messages received from other systems and occurrence of specific hardware or software conditions (operational state transitions, timer expiration, etc.). Call-back functions can also modify model entities (causing further call-back invocations) and trigger other side-effects (for example send a message to a managed end-point).

Consider, for instance, the scenario in which a user types a CLI command to shut down a port. In a traditional “procedural” system, the command triggers a pre-defined chain of function calls (and possibly IPC messages) that eventually trickles down to the port driver that executes the command. In a model-driven system, the command results in the invocation of generic code that mutates the administrative property of the model entity representing the port. The property mutation triggers the invocation of a call-back function that in turn calls the port driver to shut down the port. In the procedural case, the developer needs to design and code the whole chain from the CLI to the port driver. In the model-driven case, the developer can focus on the specific issue of reflecting administrative link state changes from the model into hardware. The rest of the functionality (validating permissions, updating state, persistifying the new configuration, logging the change, etc.) can be performed by the framework using generic code. In short, clear separation of state and behavior allows developers to focus on the logic required to handle mutations of model entities, so that actual system configuration (operational properties) converges towards requested configuration (administrative properties).

In more complex scenarios the advantages of model-driven development become even more compelling. Domain logic is clearly expressed in terms of state and behavior and strictly separated from infrastructure code. System-wide concerns such as high-availability (HA) can be factored out and dealt with at the framework level, independently from individual features. Feature interactions can be analyzed and controlled at the model level. Consolidated state for the whole system can easily be inspected at run-time and exposed to user through APIs, offering an unprecedented level of flexibility and extensibility.

## Management Principles

The framework is agnostic about the nature of the systems being managed. However, it promotes a set of principles that proved to be very valuable in building smart, flexible and highly-usable management systems.

User does not configure logical and physical system resources directly but rather defines logical (hardware independent) configurations and policies that control different aspects of system behavior. Logical configurations are rendered into concrete configurations by applying the policies in relation to the available physical resources, taking into account their state and capabilities. Concrete configurations are deployed to the managed end-points in an asynchronous, non-blocking fashion. Failed deployments are automatically retried until the operational state of the system match the desired administrative state or the error is deemed unrecoverable. Higher-level logical constructs such as templates and pools (Section 2.2) help organize configurations and resources in a logical way, avoiding error-prone, highly-repetitive tasks.

Consider for example the case in which a new hardware resource comes online. System automatically discovers it and, based on resource identity and capability, it may decide to apply a pre-provisioned configuration or to apply a new configuration generated from a template (retrieving logical resources like identifiers, addresses, etc. from pre-programmed pools) or just to acknowledge the resource and make it available for future use.

The framework is event-driven. Different types of events such as user actions, API invocations, specific hardware and software conditions are all treated uniformly.

## Framework components

The main components of the framework are:

* An XML-based language used to describe the Information Model (IM) of the managed system
* A base information model containing primitive types and useful abstractions that can be extended and specialized for specific managed systems
* A code generator (NGEN) capable of parsing the entire information model and generate various artifacts (code, schema definitions, documentation, etc.)
* A set of libraries providing:
  + core system functionalities (object manipulation, method invocation, FSM management, etc.)
  + generic system services and OS abstractions (threading, inter-process communication, etc.)
* A Data Management Engine (DME) that organizes run-time management information in a Management Information Tree (MIT) and provides API-based, transactional, role-based access to managed objects. A system developed using the framework can comprise multiple DME-based components.

The following entities are auto-generated by NGEN starting from the information model:

* APIs for retrieving, accessing and mutating managed objects
* Methods to serialize and de-serialize managed objects
* Metadata and API to examine the type and properties of managed objects at run-time
* Customizable handlers (call-backs) for events such as object mutation, method invocation, timer expiration, FSM state transition
* CLI and GUI parts
* HTML Documentation
* Information Model schema

The core architectural principle of the framework is to maintain strict separation among:

* The domain-specific content being managed, both in terms of objects and behaviors
* The platform on top of which it runs
* The applications that puts it all together

A high-level view of the framework is shown in Figure 1.

## Languages, libraries and technologies

All code (manually implemented and automatically generated) that gets executed at run-time on the back-end of the management system is written in C++. The choice of the language is primarily motivated by performance and flexibility considerations. Moreover, the C++ object-oriented, polymorphic type system is a good match for the framework object model (Section 2.1).

XML is the language of choice for the information model and for the external API. This implies that CLI, GUI and any other user interface can be coded using any language as long as it generates proper XML API calls.

Applications

Content

Platform

Manual Code

Behaviors

Generated

Objects

Methods

Behaviors

Rules

Meta

NGEN

IM

Policy Mgr

EndPoint DB

XML API

Config Mgr

Data

Management

MIT

Objects

Transactions

Behaviors

Rules

Methods

Services

Internal

APIs

External

APIs

OS Abstraction

Threads

Streams

Primitives

Communications

OS

Bindings

LINUX

Marsh.

Language

Bindings

C++

M

E

T

A

GUI

CLI

Java

3rd party  
libraries

Boost

STL

Figure 1: Insieme Framework Components

Finally, NGEN is written in Java. Modifications to NGEN are only required when the framework itself needs to be extended to support new features. Developers building applications that use the current feature set do not need any knowledge of NGEN internals.

NGEN has a strict separation between the model parsing, meta-model construction and code-generation phases. It can be easily adapted to accept as input models described in languages other than XML and to produce output code in languages other than C++.

**<TBD> Add material on Boost: what and how we use**

# Fundamental Concepts

## Object Model

### Managed Objects

The state of the managed system is represented by a collection of Managed Object (MO) instances. MOs are constructs defined in the information model, conceptually similar to a “class” in object-oriented languages. They are identified by a name and contain a set of scalar, typed values named “properties”. An MO can inherit from another MO and override (redefine) inherited properties. Only single inheritance is supported, but an MO can implement interfaces. MOs that are base classes for other MOs cannot be concrete. In other words, only the leaves of the MO inheritance tree can be instantiated.

### Management Information Tree

MO instances can contain other MO instances, forming a parent-child relationship. At run-time, all MO instances are organized in a single tree structure named Management Information Tree (MIT), managed by DME. Each MO instance is identified within the scope of its parent by a unique Relative Name (RN) and globally by a Distinguished Name (DN). The DN of a node is formed by the ordered concatenation of the RNs of the node ancestors up to the root of the MIT, using “/” as a separator. In the example of Figure 2, the RN for the bolded node is “Port-7” and its DN is “/Root/Switch-3/Linecard-1/Port-7”.

Object References Figure 2: Example of MIT, showing only node RNs

Root

Switch-3

Linecard-1

**Port-7**

Switch-2

### Object Relationships

References between MO instances that do not share a containment (parent-child) relationship are expressed through relationship definitions in the object model. This can be used, for example, to define that instances of a MO class are dependent on instances of another class.

Relationship definitions allow the framework to generically track object inter-dependencies, which can assist with fault correlation and determining object health / status. The object model definition consists of source (from) and to (target) classes, along with cardinality and exclusivity rules.

NGEN auto-generates MO class definitions that represent the source and target relationship.

The source relationship MO is contained by the FROM object. Its MO class is named:

{SOURCE MO PKG}::Rs{RELATION NAME}

The target relationship MO is contained by the TO object. Its MO class is named:

{SOURCE MO PKG}::Rt{RELATION NAME}

At runtime, a reference is created by instantiating a source relationship MO and adding it to the MIT. This can happen via an explicit request (northbound API / CLI), or implicitly in DME code.

When an instance of source relationship MO is created, the DME framework implicitly instantiates a target relationship MO when the target exists, and subsequently controls its attributes and lifecycle.

Relationships can either be explicit or named. This is specified in the XML model definition.

#### Explicit Reference

An explicit reference defines a relationship between a source MO and a specific instance of a target MO. The target instance is identified by a target DN (tDn) property that is explicitly set in the relationship source (Rs) MO.

#### Named Reference

A named reference defines a relationship between a source MO and a named target MO. The target name is derived from the naming properties of the target MO. The relationship source MO contains target name properties (each starting with ‘tn’ prefix) which are used to identify the target. Unlike an explicit reference (in which tDn will always correspond to a single unique target instance), a named reference may have multiple matching targets. The framework will attempt to form a relation with the closest matching target. The framework will implicitly set tDn with the DN of the closest matching target. Named references are primarily used when referring to policies. For details about policy resolution via named references, refer to the *Policy Manager Design Spec*.

#### Relationship State

The relationship source MO contains a ‘state’ property which is updated by the framework to reflect the state of the relationship.

Values include:

* Unformed – target MO does not exist
* Formed – target MO (and relation target) exist

Whenever relationship state changes, a behavior callback (relChangeCb()) is invoked, with both the source and relationship source (Rs) MOs passed as parameters. This method can be overridden in behavior (BI) subclasses for application-specific handling.

virtual void relChangeCb(

mo::Mo& aInMo,

mo::Mo& aInRelToMo

)

#### Relationship Model Attributes

Default values indicated by ‘\*’

**Type:**

* Named
* Explicit\*

**Scope:**

* Local\* – source and target MOs exist in the same DME process instance (in the same fabric node)
* Global – source and target MOs exist in different fabric nodes

**Cardinality:**

* 1-to-1
* n-to-1\*
* n-to-m
* 1-to-n

**Source Exclusivity:**

* Per Relation\*
* Total

**Target Exclusivity:**

* Per Relation\*
* Total

**Owner:**

* Management\*
* Oper

**Mod:**

* Explicit\* - If owner=management, then relationship can be administratively (via northbound API) modified
* Implicit – Relation is modified implicitly in DME

**Chunk Owner: Which svc / process owns primary chunk**

* Inherited from source object by default

Model Syntax:

<rel-def name="RELATIONSHIP-NAME"

from="{CLASS-NAME}"

to="{CLASS-NAME}"

cardinality="1-to-1,\_n-to-1\_,n-to-m,1-to-n"

source-exclusivity="\_per-relation\_|total"

target-exclusivity="\_per-relation\_|total"

owner="management|oper"

mod="implicit|explicit"

chunk-owner=”…”

/>

### Property attributes

Administrative properties represent the desired state of the system. Typically they can be directly modified by user and are persisted (preserved across system restarts). Operational properties represent current system state. They are generally not modifiable the user and may not be persisted, as their value can be determined at any time by querying physical resources.

A property is “explicit” if it can be changed directly by an API call, “implicit” if it can only by mutated as a side-effect of some other event.

Note that these attributes are orthogonal: both administrative and operational properties can be implicit or explicit. For example a property holding a configuration parameter that is computed by rendering a policy is administrative but implicit. A property holding the value of a counter reported by a managed end-point is operational but explicit.

### Packages

The information model supports grouping of heterogeneous constructs (primitive type definitions, MOs, methods, etc.) into “packages”. This provides modularity at the model level and prevents naming clashes. IM packages map directly to C++ namespaces in auto-generated code.

### Behaviors

The framework maintains a clear separation between state and behavior. MOs that need customized behavior are explicitly flagged in the information model. For each MO (not MO instance!) the framework generates a C++ class called “BI” (Behavior Implementation) that is distinct from the MO class. A BI class has a set of overridable methods that get invoked by the framework at well-known points in time, such as when an instance of the corresponding MO is modified, added to the MIT or deleted. Developers can fully tailor BI classes to their needs by adding custom methods and data members. However, data that is specific to individual MO instances cannot be put in BI classes; it must be explicitly added to the information model as MO properties.

### Metadata

During the code-generation phase the framework creates meta-data and APIs to allow run-time inspection of the information model. Any process with access to this data can retrieve the type and name of any property, find the base class for an MO and all MOs inheriting from it, determine if an MO can contain or be contained by other MOs.

Introspection capabilities significantly boost the power of the framework as they allow developers to write generic code that works with any MO instance, avoiding code duplication and scattering.

## Building blocks for management systems

This section describes higher-level abstractions included in the framework base information model. They are applicable to any management system and support the ideas expressed in Section 1.2.

### Organizations

An organization is a container for a set of related logical elements like policies, resource pools, templates, etc. It is meant to represent a logical partition of the user base of the system, such as a department, business unit or tenant.

Each organization can contain multiple sub-organization, thus forming a tree hierarchy that provides the scope for the resolution of logical elements and the sharing of resources. For example, a policy defined within a given organization can refer to all pools defined within the same organization or any other organization along the path to the root organization.

### Policies

Policies are named entities that control some aspect of system behavior. They are referenced by other logical elements by name (NOT by DN) and are resolved at run-time using a hierarchical scheme. A policy with the given type and name is searched starting from the hierarchy node (an organization, for example) to which the consuming element is attached and walking the tree all the way to the root. The scheme allows simple policy extension and overriding. An example is shown in Figure 3 [IFC-FS]; “EP N” is the consuming entity.

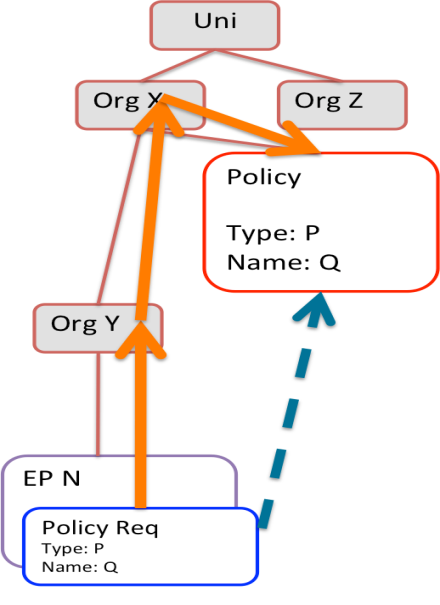


Figure 3: Example of policy resolution

### Pools

A pool is a collection of homogeneous physical or logical resources, like switch ports or MAC addresses. They can be populated manually by user or automatically by the system, as new resources are discovered.

A given resource can be a member of multiple pools at the same time but it becomes unavailable as soon as it is checked out from one pool; once it is checked back in, it becomes available again in all pools. Check-out/check-in operations can be explicitly initiated by user or can result from the application of provisioning policies.

Like policies, pools are referenced by name and follow a hierarchical resolution scheme. If a pool with available resources is not found at the level of the hierarchy where the consuming entity is attached, a pool with the same name and available resources is searched at higher levels of the hierarchy along the path to the root. This allows user to easily specify which resources are dedicated to specific entities and which can be shared. Using this scheme, dedicated resources are always used first and shared resources are tapped only if no dedicated resources are available.

### Templates

Templates are blueprints for the generation of multiple instances of the same logical configuration. Consider for example the situation in which N physical ports must be configured with identical parameters. Rather than creating N configurations, user can create a single template with the desired parameters and ask the system to instantiate it N times. The result consists of N concrete configurations that can be applied to the ports.

Templates become much more powerful when used in conjunction with pools. Imagine, for instance, that the N physical ports in the example above each need a unique MAC address. User can reference a pre-populated pool of MAC addresses from the template. Every time the system creates a concrete configuration from the template, it automatically retrieves an address from the pool and uses it. When the instance is deleted, the address is automatically returned to the pool.

### Faults

Faults are explicitly represented in the run-time model as MO instances. They are automatically instantiated, escalated, de-escalated and cleared by the system when specific conditions are detected. The lifecycle of faults is controlled by fault policies. See [IFC-FM] for further details.

### Capability Catalog

The capability catalog is a part of the information model that contains a description of all known hardware resources and the capabilities they support. Feature requirements are expressed in terms of generic resource capabilities rather than specific hardware parts.

The catalog can be updated at run-time, allowing the integration of new hardware into a running system without software upgrades and system downtime.

#### Catalog

Capability Catalogue provides an abstraction of feature capabilities. The MIT contains a single catalogue, which contains sub-trees of catalogue entries (capability providers). The catalogue is loaded and persisted during initial IFC bring-up. In addition, IFC supports a mechanism to update / reload catalogues at runtime. Catalogue updates allow existing software to manage future hardware and software revisions.

Application code queries the capability catalog to determine supported features on a given type / version of hardware and software.

#### Provider

Capability providers define a set of capabilities (capability definitions) specific to a type / version of hardware. There is an abstract base class (capabilityProvider) from which all providers are subclassed. Provider entries are indexed by a 3-tuple key consisting of vendor, model, and revision.

#### Def

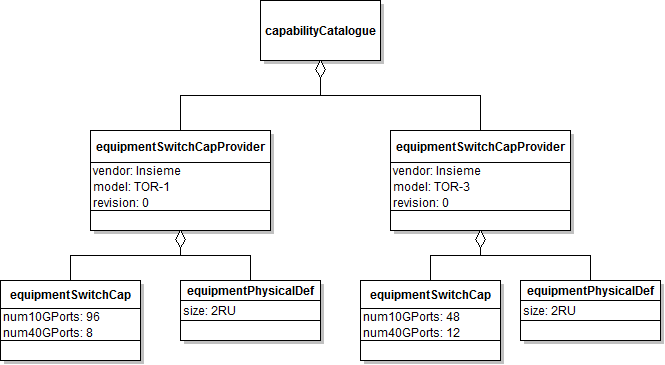
Capability definitions define a semantically similar set of capabilities. There is an abstract base class (capabilityDef) from which all definitions are subclassed.

#### Capability Base Classes Object Model

#### 

#### Capability Example

In fabric nodes, physical characteristics such as chassis size and number of ports can be modeled as capabilities. Example model:



# Application Programming Interface

External components such as UIs, scripts and third-party orchestration software interact with DME using a socket-based northbound interface. Clients send request messages called *stimuli* that contain API method invocations together with the data needed to serve the request (method parameters, user credentials, etc.). Each stimulus is processed by DME in the context of a transaction and generate a response that is sent back to the client (Figure 4). The response indicates the transaction success or failure result and optionally the affected object subtree.

**Communication  
Channel**

**DME-based**

**process**

**Changer**

**MIT**

**MOs**

**External**

**Request**

**Stimulus**

**api**

**request**

**api**

**response**

**Managed End Point**

**async**

**object**

**change**

**notification**

**async deployment  
(not in transaction)**

**Transactor**

**Doer**

**XML Interface**

A transaction fails if any aspect of the operation cannot be fully validated by the model (arguments out of range, known lack of resources, insufficient rights, …). Failed transactions leave the MIT state unchanged. Note that transaction results only refer to the application of stimuli to the model. Failures to deploy configurations to managed end-points are not reflected in the response message, as deployments happen asynchronously, outside the scope of the transaction. User is informed of failed deployments by means of faults and associated notification mechanisms (syslog, call-home messages, etc. ).

The northbound interface also supports an asynchronous event notification mechanism that allows clients to be informed of changes posted to the MIT (Section 3.3) without having to poll.

Figure 4: API control flow

## API Methods

The core of the API are the *resolve* and *config* methods, which allow client to perform CRUD (create, read, update, delete) operations on the MIT.

All read query functionality is provided via a single Resolve method. This method supports:

* Resolve by DN: find MO instance with given DN
* Resolve Children: find all immediate children for a given MO instance
* Resolve Subtree: find all children and their subtrees for a given MO instance
* Resolve Class: find all MO instances of the specified class
* Scope: limit the scope of “resolve” methods, within the transaction, to a particular sub-tree

Resolve methods generally accept the option to return individual MO instances matching the search criteria or the instances together with all their descendants. They also accept property filters to make the search process more selective.

Config methods include:

* ConfMo: create, modify or delete a single MO instance
* ConfMos: create, modify or delete a multiple MO instances in a single transaction
* EstimateImpact: do a “dry run” of a config change and return the complete set of affected MO instances, without actually applying the changes

Config methods can also work with subtrees and accept property filters.

A DME-based process may also provide custom methods for login/logout, policy resolution, forced fail-over, etc.

DME’s hierarchical object model approach is a very good fit for RESTful interfaces. [IFC-FS] provides an example of RESTful API built on top of the base DME API.

## Stimuli and Transactions

Access to MIT through the external API happens in a transactional way, meaning that either all the changes within a transaction are applied to the model or none is applied. Transaction boundaries are implicitly defined by DME and cannot be controlled by clients. A transaction begins when DME starts processing a stimulus and ends once all explicit and implicit (via call-back functions) model changes have been validated, applied to the MIT and made persistent. After the end of the transaction, side-effects (including client notifications and deployment to managed end-points) are made visible.

DME processes stimuli serially. Pending stimuli are inserted into a priority queue. Higher-priority stimuli are processed first. This allows stimuli that dictate administrative changes to execute before stimuli that provide asynchronous operational updates, making the management system more responsive and UIs more snappy. Stimuli are processed in FIFO order within a given priority class.

A stimulus is always processed in the context of a single transaction; it never spans multiple transactions. In order to minimize transaction overhead, DME attempts to process multiple “bulkable” stimuli within a single transaction. If any stimulus causes a failure, DME rolls back the entire transaction and re-processes the stimuli one at a time. This step preserves stimulus semantics in terms of transaction boundaries. This optimization not only reduces the processing overhead of the stimuli but also of the side-effects. For example, object change notifications can typically be bulked and sent to clients in a single message.

While DME is making the side-effects of a transaction persistent, it starts working on the next transaction. This achieves a pipe-lining effect that increases throughput by avoiding idle time due to I/O and possibly performing the two operations in parallel. The two transactions do not clash because the old one does not need to modify the MIT to achieve persistence and the side-effects of the new one are only made visible until the old one is persistified.

### ACID properties

The DME transactor guarantees that the ACID (Atomicity, Consistency, Isolation, and Durability) properties are satisfied for each transaction.

#### Atomicity

Atomicity requires that each transaction is "all or nothing": if one part of the transaction fails, the entire transaction fails, and the database state is left unchanged. DME achieves this by keeping a log of all the changes performed explicitly or implicitly to the MIT during a transaction. In case of failure the change log is used to roll-back the entire transaction and restore the original state of the MIT.

#### Consistency

Consistency requires that data-base integrity constraints are respected before and after each transaction. This is achieved by subjecting all explicit and implicit data modifications to rigorous model-level validation (property-level range checking, object-level semantic checking, relationship-level checks, access permissions checks, etc.).

#### Isolation

Isolation requires that a transactions does not see the partial results of other transactions executed in parallel. The constraint is satisfied trivially by executing transactions serially. This avoids altogether the extremely complex issues related to concurrent execution of transactions.

#### Durability

Durability ensures the persistence of the data model in the presence of failures. This includes software failures as well as hardware failures. DME achieves persistence by keeping an up-to-date copy of the MIT on secondary storage. A transaction is not considered to be committed until data has been written to secondary storage. In case of power failure, scheduled reboot or software crash, DME reloads the MIT from secondary storage and recovers system state up to the last committed transaction. For performance reasons, some portions of the MIT may not be persisted.

### Other API properties: idempotency, forgiveness

Idempotency is the property by which invoking the same operation repeatedly does not change the result beyond the first invocation. In framework terms, if the same stimulus is applied to the model N times in a row, it should produce side-effects only the first time.

Idempotency dramatically simplifies the interaction between nodes in a distributed environment. Consider for instance a client interacting with a remote DME-based system. If the client sends a request to the DME-based system and doesn’t receive a response, it can just keep retrying following whatever strategy it prefers. It doesn’t have to ponder whether the system is unreachable or just busy. It doesn’t need to keep a retry count. As soon as it receives a response, it knows that the changes in the request have been applied. If multiple copies of the message were delivered, in any order, idempotency nullifies the effects of all but the first one. It is much easier to make sure that at least one copy, rather than exactly one copy, of a message was delivered.

Forgiveness is the idea that when the system is exposed to some unexpected condition, it should handle it gracefully and liberally, but always preserving the consistency of the model and the integrity of the system. For example, if DME finds an MO instance of unknown type while loading the MIT at boot time, it should just skip it. If system receives a request to modify an unknown property of an MO, it should just ignore it. The forgiveness principle is useful in general because it improves the robustness of the software. However, it is essential to guarantee backward/forward compatibility in upgrade/downgrade scenarios, including temporary version mismatches between different software components. This is a common scenario that occurs when redundant systems are upgraded in steps to avoid downtime.

Note that idempotency and forgiveness cannot be guaranteed by the framework alone. The system exhibits these properties and delivers the corresponding benefits only if all parts, including custom behavior code, satisfy them. For example a call-back function that creates an MO instance as a side effect should check if the MO instance is already there and do not re-create it if it is. Creating a new one every time would break the idempotency. A forgiving call-back function that expects an MO instance should not crash if the MO instance is not there but rather create it at the beginning and then execute normally.

## Generic Object Notifications

The framework supports a generic object notification mechanism that allows API clients to receive notifications for any change posted to the MIT. Using this mechanism clients can build local caches of MO instances and maintain coherence with MIT without ever polling for updates.

Notifications are produced for all MOs, except those that are explicitly flagged as “non-reportable” in the information model. Individual clients can specify filters based on MO class and/or subtree.

Notifications are produced at the end of each transaction, and are released to the northbound adapters after the transaction is committed. There is one notification per object that changes.

Each notification message specifies:

* Type of notification: object creation, object deletion, or object modification
* The object’s DN and class for identification. (The class information is necessary only for object creation.)
* The list of modified properties, together with the most recent values for each property. Note that in the case where an object’s property value is changed multiple times during a single transaction, only a single object change notification message is generated and only the most recent value will be provided.
* Time stamp (UTC)
* Transaction identifier.
* Cause of change: administrative or operational. This is useful for filtering purposes to know which changes were done via a northbound interface.
* Global Sequence Number. This counter is initialized with the DME when the DME starts and increments with each object change notification message. This sequence number is used by the client to identify any gaps in the stream.

Further details can be found in [IFC-EED].

# References

[IFC-EED] IFC External Event Dispatcher Functional Specification

[IFC-FM] IFC Fault and Event Management Functional Specification

[IFC-FS] IFC Overall Functional Specification

[SAM-FS] SAM Overall FS – EDCS-726493

[UCSM-DEV] UCS Manager Developer’s Guide – EDCS-849998