İSTANBUL TEKNİK ÜNİVERSİTESİ ★ FEN BİLİMLERİ ENSTİTÜSÜ

HLA NESNE MODEL ŞABLONU TEMELLİ FOM-ÇEVİK KOD ÜRETİMİ

YÜKSEK LİSANS TEZİ

Cemil AKDEMİR

Anabilim Dalı : Bilgisayar Mühendisliği

Programı : Bilgisayar Mühendisliği

HAZİRAN 2009

İSTANBUL TEKNİK ÜNİVERSİTESİ ★ FEN BİLİMLERİ ENSTİTÜSÜ

HLA NESNE MODEL ŞABLONU TEMELLİ FOM-ÇEVİK KOD ÜRETİMİ

YÜKSEK LİSANS TEZİ
Cemil AKDEMİR
(504061507)

Tezin Enstitüye Verildiği Tarih : 04 Mayıs 2009
Tezin Savunulduğu Tarih : 03 Haziran 2009

Tez Danışmanı : Prof. Dr. Nadia ERDOĞAN (İTÜ)
Diğer Jüri Üyeleri : Yrd. Doç. Dr. Şima ETANER UYAR (İTÜ)
Yrd. Doç. Dr. Yunus Emre SELÇUK (YTÜ)

HAZİRAN 2009

Aileme,

ÖNSÖZ

Bu çalışmamın ortaya çıkmasında emeği bulunan başta ailem olmak üzere, tez danışmanım sayın Prof. Dr. Nadia Erdoğan'a, gerekli bilgi birikimini edinmemde her türlü desteği veren ve tezimi sonuçlandırmam hususunda özverili davranışlarından ötürü arkadaşlarıma teşekkür ederim. Çalışmamın bu alanda yapılacak ileriki çalışmalara ışık tutmasını temenni ederim.

Nisan 2009

Cemil Akdemir

(Bilgisayar Mühendisi)

İÇİNDEKİLER

Sayfa

KISALTMALAR

API : Application Programming Interface
CGF : Computer Generated Forces
DARPA : Defence Advanced Research Project Agency
DIS : Distributed Interactive Simulation
ERD : Entity Relationship Diagram
FEDEP : Federation Development and Execution Process
FOM : Federation Object Model
FYA : Federe Yönetim Altyapısı
GenDL : GENESIS Description Language
HLA : High Level Architecture
IEEE : Institute of Electrical and Electronics Engineers
LOC : Line of Code
MOM : Management Object Model
M&S : Modeling and Simulation
NATO : North Atlantic Treaty Organization
OMT : Object Model Template
ONERA : Office National d'Études et de Recherches Aérospatiales
PDL : Program Design Language
RPR : Real-time Platform Reference
RTI : Run Time Infrastructure
RTIME : Run Time Infrastructure Modeling Extensions
SISO : Simulation Interoperability Standards Organization
SOM : Simulation Object Model
SQL : Structured Query Language
STANAG : NATO Standardization Agreement
TENA : Test and Training Enabling Architecture
UML : Unified Modeling Language
XML : Extensible Markup Language

ÇİZELGE LİSTESİ

Sayfa

ŞEKİL LİSTESİ

Sayfa

HLA NESNE MODEL ŞABLONU TEMELLİ FOM-ÇEVİK KOD ÜRETİMİ

ÖZET

Simülasyon sistemleri, başta askeri olmak üzere özellikle karmaşık, gerçeklenmesi zaman alıcı ve pahalı sistemlerin modellenmesi, eğitim etkinliğinin arttırılması ve performans değerlendirmesi gibi birçok alanda kullanılmaktadır. Bu nedenle, sistemlerin birlikte çalışabilirliğinin önemi giderek artmaktadır. Benzer alanlarda daha önce yapılmış ürünlerin ya da sistemin geneline hizmet edebilecek düzeyde tasarlanmış bileşenlerin, yeni sistemlerde kullanılabilmesi istenmektedir. Bu ihtiyacın sağlanması, bileşenlerin geliştirilmesinde ortak bir anlayışın oluşmasını zorunlu kılmıştır.

Bu yöndeki ilk çalışmalar 1980'li yılların sonunda gerçek-zamanlı silah sistemlerinin geliştirilmesi amacıyla DARPA (Defence Advanced Research Project Agency) tarafından SIMNET adı altında başlatılmıştır. 1990'lı yılların başından itibaren etkileşimli dağıtık simülasyon sistemlerinin birlikte çalışabilmesini öngören Dağıtık Etkileşimli Simülasyon (Distributed Interactive Simulations-DIS) çalışmaları yine DARPA öncülüğünde yürütülmüştür. Bu çalışmalar, 1995 yılında NATO tarafından yayınlanan STANAG 4482 anlaşması ile bir standart olarak kabul edilmiştir. DIS, aynı zamanda IEEE tarafından 1278 Standardı ile tanımlanmıştır. 1996 yılında, ilk çıkışı itibari ile DIS standardının bir sonraki adımı olarak görülen ve DIS++ olarak isimlendirilen Yüksek Seviyeli Mimari (High Level Architecture-HLA) çalışmaları başlatılmış (HLA 1.3) ve 2000 yılında bu çalışmalar IEEE tarafından 1516 standardı (HLA1516) olarak tanımlanmıştır.

Simülasyon sistemleri için yapılan bu ortak mimari çalışmaları, iletişimde kullanılacak olan verilerin de belirli standartlarda tanımlanması ihtiyacını doğurmuştur. Özellikle HLA bir standart olarak kabul edildikten sonra, IEEE 1516.2-2000 - Standard for Modeling and Simulation High Level Architecture - Object Model Template (OMT) ile bu verilerin tanımlamaları standart olarak belirlenmiştir. Bundan sonra, SISO (Simulation Interoperability Standards Organization) tarafından bu amaçla kullanılabilecek şablon nesne modelleri tanımlanmıştır.

Bu çalışma, tanımlanmış şablonlar kullanılarak geliştirilecek olan dağıtık simülasyon sistemlerinin ortak kullanabileceği veri iletişim katmanının kod üretme teknikleri kullanılarak üretilmesini ele almıştır. Aynı zamanda, katmanın belirlenmiş standartlara uygun oluşturulması amaçlanmıştır. Bunun yanında hem geliştirilecek bileşenlerin simülasyon bağlantılarının sağlanacağı bir alt katman hem de uygulama kodunun geliştirilebileceği başka bir katman da üretilen katmanla entegre edilerek genel bir simülasyon bileşeni oluşturma altyapısı sunulmuştur. Geliştirilen altyapı, sistemlerin ihtiyaçları doğrultusunda ortaya çıkabilecek nesne modeli değişiklerinden geliştirilen uygulamaların en az etkilenmesi amacıyla FOM (Federation Object Model)-çevik geliştirilmiştir. Altyapı üzerinde geliştirilen uygulamaların mevcut sistemlerle uygunluğu ve benzer katman yazılımları ile performans karşılaştırılmasına da değinilmiştir.

HLA OBJECT MODEL TEMPLATE BASED FOM-AGILE CODE GENERATION

SUMMARY

The interoperability of simulations systems has incredibly become important after their wide usage including mostly military purposes, time-consuming and expensive systems, incrementing the effectiveness of education systems and performance analysis. The idea of reusability of the components that were previously designed for similar systems or the components that can serve within multiple system, made it necessary to have a common understanding in developing such systems.

Primary works in this area were started by DARPA under the name SIMNET, for the purpose of real-time weapon system development in late 1980s. From the beginning of 1990s, DARPA again led the research about making Distributed Interactive Simulations possible. These research yielded a NATO standard in 1995 by the STANAG 4482 agreement. Meanwhile, DIS is defined as 1278 Standard by IEEE. In 1996, High Level Architecture, firstly-named as DIS++ because of being defined as the next step of DIS, was defined in HLA 1.3 and eventually standardized in HLA1516 by IEEE.

All this effort for the name of developing common architecture for simulation systems, also caused a new requirement to define the data that is used in communication with respect to some other standard. Especially, after HLA was agreed as a standard, the definition of these data structures was also standardized with IEEE 1516.2-2000 Standard for Modeling and Simulation High Level Architecture - Object Model Template (OMT) by IEEE. Starting after that, object model templates are also defined for this purpose, led by SISO.

This work is intended to develop an auto-generated data communication layer from object model templates. The layer is supposed to be compatible with existing standards and can commonly be used by distributed simulation components. This layer is integrated with two additional layers within the developed framework, one for linking the application with the simulation and the other for developing the domain code onto it. The framework also designed as FOM-agile to keep the developed applications minimally affected from the changes in object model. The applications over the framework are also tested with existing applications from the point of view of compatibility, usage and performance.

1. GİRİŞ

Çalışmanın ilk bölümünde, HLA ile ilgili temel kavram ve bileşenler tanıtılmış olup bu alanda yapılmış benzer çalışmaların kapsam ve uygulamalarına yönelik değerlendirilmeleri yapılmıştır. Her çalışma sağladığı yenilikler ve getirdiği kısıtlar açısından ele alınmıştır.

İkinci bölümde dağıtık simülasyon sistemlerine ve kullanılan mimarilere temel bakışla birlikte, çalışmanın uygulama alanı olan ve IEEE STD 1516.2-2000 ile belirlenmiş olan HLA nesne model şablonu hakkında detaylı bir inceleme sunulmuştur.

Kod üretim teknikleri, uygulama alanları, sağladığı katkılar ve uygulanan metotlara ilişkin detaylar verildikten sonra, nesne model şablonu temelli kod üretiminin bu teknikler kullanılarak gerçeklenebilirliği üzerine çalışmalar üçüncü bölümde ele alınmıştır.

Dördüncü bölüm, yapılan çalışmanın tasarımı ve gerçeklenmesi ile ilgili detaylı bilgiler içermekle birlikte geliştirilen altyapının performans, uyumluluk ve etkinlik açısından benzer çalışmalarla kıyaslanması da bu bölümde ele alınmıştır.

Sonuç bölümünde geliştirilen altyapı ve bu altyapının ileriki çalışmalar kapsamında ne şekilde geliştirilebileceği değerlendirilmiştir.

1.1 Tezin Amacı

Bu çalışmada HLA standardında belirtildiği şekli ile simülasyon arakatman yazılımının otomatik kod üretme teknikleri ile oluşturulması konusu incelenmiştir. Mevcut çalışmaların genel olarak askeri projeler kapsamında ve ihtiyaçlar doğrultusunda geliştirilmiş olması bu yazılımların kolay elde edilebilmesini ya da değiştirilerek kullanılabilmesini mümkün kılmamaktadır.

Diğer taraftan farklı nesne modelleri üzerine geliştirilecek yazılımların olabildiğince FOM-çevik tasarlanması ancak ticari ürünlerle sağlanmıştır. Bu ürünlerin lisans maliyetleri göz önüne alındığında, bunların büyük ölçekli projelerde kullanımının maliyeti ve bakımı önemli ölçüde zorlayacağı düşünülmektedir. Bu nedenle çalışmada, ihtiyaçlar doğrultusunda şablon temelli değiştirilebilecek ve geliştirilen uygulamayı nesne model değişikliklerinden mümkün olduğunca yalıtacak bir yazılım altyapısının tasarım ve geliştirilmesi ele alınmıştır.

1.2 HLA Kavramları

Çalışma boyunca kullanılacak olan HLA kavramlarına ilişkin açıklamalar aşağıda verilmiştir:

- *Federe*: HLA uyumlu olarak gerçekleştirilmiş ve dağıtık olarak çalışabilen simülatör uygulamalarıdır.
- *Federasyon*: Etkileşim içinde bulunan bir grup federenin biraraya gelerek oluşturduğu sistemdir.
- *Koşum Zamanı Altyapısı (Runtime Infrastructure-RTI)*: HLA üzerinde geliştirilmiş uygulamalar(federeler) için ortak arayüz servislerini sağlayan yazılımdır.
- *Nesne Sınıfı*: Federeler arasında veri alış verişinde kullanılan ve belirli bir grup özelliği (attribute) içeren simülasyon nesneleri olarak tanımlanırlar.
    - Kalıcı özelliktedirler. Federeler tarafından simülasyon ortamına kayıt ettirilip, güncellenebilme ve silinebilme özelliklerine sahiptirler.
    - İçerdikleri özellikler veri alanlarından oluşmakta olup herhangi bir fonksiyonellik taşımazlar.
- *Özellik*: Nesne sınıflarını oluşturan veri alanlarının her birine verilen addır.
- *Etkileşim Sınıfı*: Federeler arasında veri alış verişinde kullanılan ve belirli bir grup parametre (parameter) içeren simülasyon mesajı olarak tanımlanırlar.
    - Geçici özelliktedirler. Federeler tarafından simülasyon ortamına belirli bir olayın duyurulması amacıyla anlık olarak atılmaktadırlar.

- İçerdikleri parametreler veri alanlarından oluşmakta olup herhangi bir fonksiyonellik taşımazlar.

- *Parametre*: Etkileşim sınıflarını oluşturan veri alanlarının her birine verilen addır.
- *Nesne Model Şablonu (Object Model Template-OMT)*: Nesne modellerinin tanımlanması amacıyla kullanılan şablonlardır. Federasyon Nesne Modeli ve Simülasyon Nesne Modellerinin tanımlanmasında kullanılır.
- *Simülasyon Nesne Modeli (SOM)*: Belirli bir federenin herhangi bir federasyonda gönderip alabileceği nesne ve etkileşim sınıflarının tanımlandığı nesne modelleridir.
- *Federasyon Nesne Modeli (FOM)*: Belirli bir federasyondaki federeler tarafından gönderilip alınabilecek nesne ve etkileşim sınıflarının tanımlandığı nesne modelleridir. Federasyona dahil olan federelerin SOM'larının birleşimi olarak düşünülebilir.

1.3 HLA Bileşenleri

HLA bileşenleri, HLA tarafından belirtilmiş olan kurallarla, koşum zamanı altyapısı tarafından sağlanması gereken servisleri ve nesne model şablonu tanımlamalarını içermektedir. Bu bileşenlerle ilgili bilgiler alt başlıklarda incelenmiştir.

1.3.1 HLA kuralları

Federasyona katılan federeler arasındaki ilişkilerle, federe ve federasyonun sorumlulukları atasındaki ayrımı belirleyen kurallardan oluşur. Beşi federasyon, beşi federeler için olmak üzere toplam on kuraldan oluşmaktadır.

Federasyon kuralları şu şekilde tanımlanmıştır:

- Her federasyon HLA OMT uyumlu bir FOM'a sahiptir.
- FOM'daki her nesnenin temsili RTI değil federeler tarafından sağlanır.
- Federasyon boyunca federeler arasındaki tüm iletişim RTI ile sağlanır.
- Federeler RTI ile HLA Interface Specification uyumlu etkileşirler.

- Federasyonda bir anda bir nesne sınıfı örneğinin özelliği ancak bir federe tarafından sahip olunabilir.

Federe kuralları ise aşağıda verilmiştir:

- Her federe HLA OMT uyumlu bir SOM'a sahiptir.
- Federeler nesne ve etkileşim gönderme ve alma işlemlerini SOM'larına uygun yapar.
- Federeler koşum esnasında nesne özelliklerinin sahipliklerini SOM'larında belirtildiği şekilde dinamik olarak değiştirebilirler.
- Üye değişken güncellemelerinin hangi koşullar altında yapılacağı SOM uyumlu gerçekleştirilir.
- Federeler, federasyondaki diğer federeler ile veri alışverişlerini koordine etmek amacıyla yerel zaman yönetimi gerçekleştirebilirler.

1.3.2 Arayüz tanımlamaları (Interface specification)

Arayüz tanımlamaları, RTI tarafından federeler arası iletişim ve federasyon için sağlanması gerekli hizmet ve servisleri tanımlamaktadır. Bu servisler ve amaçları aşağıdaki şekilde özetlenebilir:

- Federasyon Yönetimi: Federasyonun oluşturulması, yok edilmesi gibi hizmetleri içermektedir.
- Gösterim Yönetimi: Yayınlanacak ve abone olunacak nesne ve etkileşimlerin belirtilmesi işlemlerini içerir.
- Nesne Yönetimi: Nesne örneklerinin kaydettirilmesi, güncelleme ve etkileşim gönderilmesi gibi işlemleri içerir.
- Sahiplik Yönetimi: Nesne özelliklerinin sahipliklerinin alınması/devredilmesi gibi hizmetlerdir.
- Veri Dağıtım Yönetimi: Abone ve Yayın alanlarının ve kesişimlerinin belirlenmesine yönelik hizmetlerdir.
- Zaman Yönetimi: Zaman Kısıtlı (Time Constraint) ve Zaman Düzenleyici (Time Regulating) federelerin koordinasyonunun sağlanmasını sağlar.

1.3.3 Nesne model şablonu (OMT)

İletişimde kullanılan veri ve federeler arasındaki etkileşim hakkında genel bir bilgi verir. Federasyona dahil olabilecek federelerin yeteneklerini tanımlamak için standart bir mekanizma sunmakla birlikte, HLA nesne modellerinin tasarlanıp üretilmesinde kullanılabilecek araçlar için tanımlamalar da içerir. Bu tanımlamalar nesne model şablonlarına ait tablolar kullanılarak gerçekleştirilir. Nesne model şablonu ve tablolarına ait detaylı bilgiler sonraki bölümde anlatılmaktadır.

1.4 Literatür Özeti

HLA'in dağıtık simülasyon sistemlerinin üzerinde çalışacağı bir mimari olarak ilk ortaya çıkışından itibaren bu mimari üzerinde geliştirilecek olan simülasyon yazılımlarının gerçeklenmesinin kolaylaştırılması ve birlikte çalışabilirliğin en üst düzeyde sağlanması amacıyla çok çeşitli çalışmalar yapılmaktadır. Bu çalışmaların en önemlisi IEEE tarafından HLA tabanlı bileşenlerin ve sistemlerin geliştirilme sürecine ait yayınlanan standartlardır:

- IEEE STD 1516-2000 Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) — Framework and Rules: HLA üzerinde geliştirilecek bileşenlerin ve sistemlerin uyması gereken 10 temel kural belirlenmiştir [1].
- IEEE STD 1516.1-2000 Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) Federate Interface Specification: HLA tarafından sağlanması gereken servislerin tanımlandığı standarttır [2].
- IEEE STD 1516.2-2000 Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) Object Model Template (OMT) Specification: Bileşen ve sistemlerin uyması gereken nesne modellerinin şekil ve dizilimlerinin belirtildiği standarttır [3].
- IEEE STD 1516.3-2000 Recommended Practice for High Level Architecture (HLA) Federation Development and Execution Process (FEDEP): HLA sistemi geliştirme ve çalıştırma sürecine ait takip edilmesi gereken adımları içerir [4].

Bu alanda yapılan diğer çalışmalar genel olarak yukarıda bahsedilen standartların bir ya da birkaçını kapsayan ve bu mimari üzerindeki bileşenlerin ortak kullanacağı ya da geliştirileceği platformların ve çerçeve yazılımlarının tasarlanması olarak özetlenebilir.

1.4.1 Koşum Zamanı Altyapısı Modelleme Uzantıları

Bu alanda ilk çalışmalardan biri Amerikan Deniz Kuvvetleri Sualtı Merkezi tarafından geliştirilen Yinelemeli Tasarım Aracı (Recursive Design Tool) adı altında sunulmuştur. Çalışmada, HLA geliştirme süreci Top-down ve Bottom-up olmak üzere iki şekilde ele alınmıştır. Top-down yaklaşımda geliştiriciler varolan FOM yapılarından yola çıkarak gereksinimleri doğrultusunda yeni eklentiler yapıp bileşenlere ait nesne modellerini türetir. Simülasyon bileşenleri daha sonra bu modellere uygun olarak gerçeklenir. Bottom-up yaklaşımda ise mevcut simülasyon bileşenlerini oluşturan nesne yönelimli modeller kullanılarak bileşene ait simülasyonda kullanılacak nesne modeli (SOM) üretilir. Son olarak tüm bileşenlerin nesne modelleri birleştirilerek, federasyon nesne modeli (FOM) üretilir. Bu araç geliştirilerek üretilen Koşum Zamanı Altyapısı Modelleme Uzantıları (Run-Time Infrastructure Modeling Extensions-RTIME) isimli araç ise Varlık İlişki Diyagramlarından (Entity Relationship Diagram-ERD) simülasyon nesnelerinin ve uygulama nesnelerinin türetilmesinin yanında uygulama nesnelerine ait durum diyagramlarının modellenmesi imkanlarını sunmaktadır. Aracın sağladığı Program Tasarım Dili (Program Design Language-PDL) yardımıyla geliştiriciler aynı zamanda üretilen koda eklenmesini istedikleri metotları da tanımlayabilmektedirler [5].

RTIME aracının HLA geliştirme sürecinde yer alan her iki yaklaşımı da desteklemesi, üretilen kodun uygulama düzeyinde işletilecek olan akışa ait durumları ve geçişleri içerecek şekilde üretilmesi ve üretilen bu şablon koda kullanıcı tanımlı metotların eklenebilmesi çalışmanın güçlü yönlerindendir. Diğer taraftan bu işlemlerin gerçekleştirilmesi amacıyla HLA geliştirme sürecinde yer almayan adımlara ihtiyaç duyulması ve üretilecek kodun araca özel bir geliştirme dilini gerektirmesi aracın yaygın kullanımına olanak vermemiştir.

1.4.2 GENESIS

Fransız Ulusal Havacılık ve Uzay Ajansı (ONERA) tarafından 2003 yılında üç yıllık bir proje olarak başlatılmıştır. Temel amacı listelenen kriterlere uygun olarak bir çerçeve geliştirmektir:

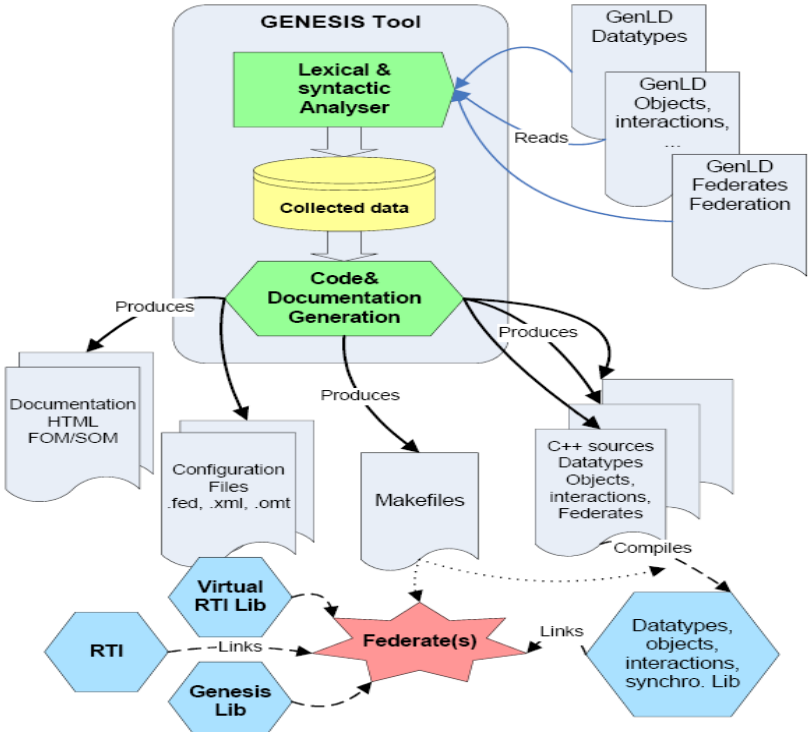
- Federe nesne modeli ile yazılım nesneleri arasında bir ilişki oluşturabilmelidir.
- Mevcut olan federe ve federasyon modellerine kolay uygulanabilmelidir.
- FEDEP sürecinin başından sonuna tüm süreçleri kapsayacak şekilde modellenmelidir.
- Ek bir geliştirme ihtiyacı olmaksızın tümüyle işlevsel federeler üretebilmelidir.

Bu isterlerin karşılanabilmesi amacıyla bu platform da GENESIS Geliştirme Dili (GENESIS Description Language-GenDL) isimli biçimsel bir dile sahiptir. Aynı zamanda IEEE standartları ile tanımlanmış kavramlara ek olarak beklentilerin karşılanması amacıyla yeni kavramlar da platforma dahil edilmiştir. Projenin çıkış noktası betimsel (declarative) bir yapıya sahip HLA kavramlarının prosedüral diller ile gerçeklenmesinde yaşanılan güçlükler olarak tanımlanmıştır ve ihtiyaçları eksiksiz olarak tanımlayabilecek bir betimsel dil kullanılması durumunda hedef dil ne olursa olsun bu sürecin otomatikleştirilebileceği dile getirilmiştir. Hedef dile ait kod üretimi bu platformda plug-in yapısı ile sağlanmış olup yeni hedef diller için genişleyebilme özelliği kazandırılmıştır [6].

Bu platformda geliştirme süreci şu şekilde özetlenebilir:

- Simülasyonda var olacak veri tipleri, etkileşim ve nesne sınıfları GenDL kullanılarak tanımlanır.
- GENESIS platformu, toplanan veri içinde sözlüksel ve dizilimsel kontrolleri yapar.
- Federasyonda kullanılacak olan FOM/SOM dokümantasyonu, hedef dile ait kaynak kod dosyaları ve derleme için gerekli dosyalar üretilir.
- Derleme dosyaları kullanılarak federelere ait çalıştırılabilir dosyalar elde edilir.

Şekil 1.1'de platformun bu akışı özetlenmiştir.

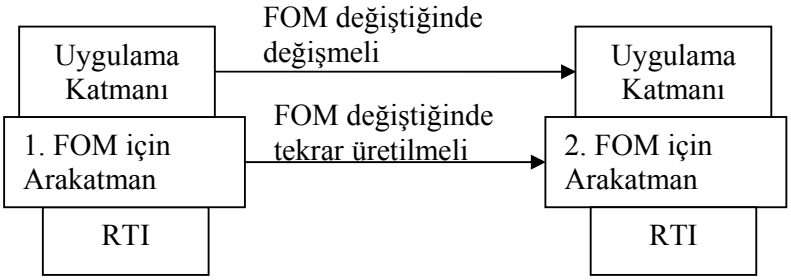


Şekil 1.1 : GENESIS platformunun yapısı [6].

GENESIS, IEEE tarafından standartta belirtilmiş özelliklerin çok büyük bir kısmını kapsaması ve hedef dil olarak genişleyebilir nitelikte olması sebebiyle bu alanda yapılmış önde gelen çalışmalardandır. Diğer taraftan projenin askeri nitelikli olması ve ihtiyaçlar doğrultusunda geliştirilmesi nedeniyle zaman yönetimi gibi performans gerektiren HLA servislerini öntanımlı olarak kullanması ve geliştirme sürecince HLA kavramları dışında kavramlar içermesi nedeniyle yaygın kullanılmamıştır. Bu platform da önceki platform gibi geliştirme süreci için kendi özel dilini kullanmaktadır.

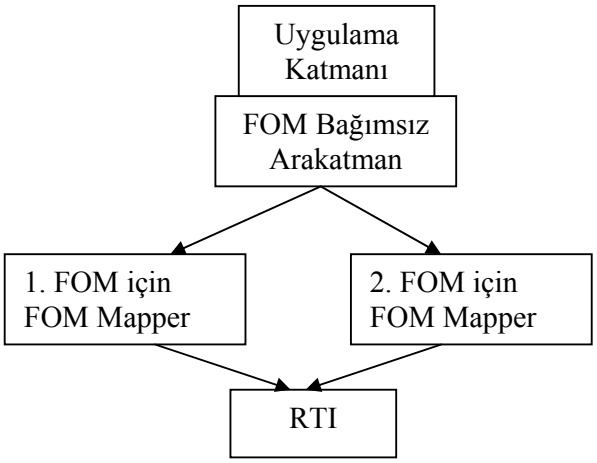
1.4.3 VR-Link

MAK firması tarafından sunulan ticari bir üründür. Çıkış noktası, bu altyapı kullanılarak geliştirilen federelerin ortaya çıkan ihtiyaçlar doğrultusundaki FOM değişikliklerinden etkilenmesinin önlenmesidir. Önceki iki çalışma FOM temelli kod üretme yaklaşımını kullandığı için FOM değişiklikleri durumunda bu altyapılar kullanılarak geliştirilen uygulamaların yeniden derlenerek üretilmesi gerekmektedir [7]. FOM temelli kod üretim mimarisi Şekil 1.2'de gösterilmiştir.



Şekil 1.2 : FOM temelli mimari.

Diğer taraftan VR-Link, bu iki mimarideki eksikliği RTI ve FOM API (Application Programming Interface) katmanları arasına eklenen bir eşleme (mapping) katmanı ile gidermiştir. Buna göre uygulama katmanında kullanılan nesneler ile simülasyon nesneleri arasındaki eşleme, eklenen bu katman tarafından yönetilmektedir. FOM değişikliği durumunda eşleme katmanına yapılan eklentilerle mevcut eşleme tabloları güncellenmekte ve uygulama kodunun tekrar derlenmesi ihtiyacı ortadan kalkmaktadır [7]. FOM bağımsız bu yaklaşım ise Şekil 1.3'de gösterilmiştir.



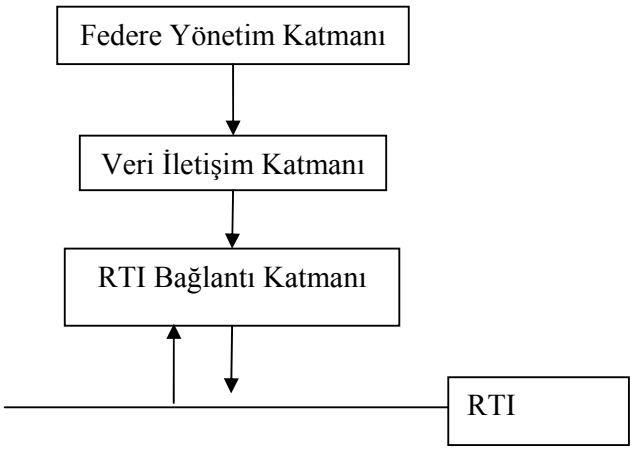
Şekil 1.3 : FOM bağımsız mimari.

Bu çalışmada, FOM eşleme katmanında yer alan kodun otomatik olarak üretilip üretilemeyeceği de irdelenmiş ve FOM sınıfları ile uygulama sınıfları arasındaki ilişkinin çok çeşitli şekillerde kurulabilmesi ve bu işlemin betimsel bir formatta tanımlanamaması sebebiyle bu işlemin pratik olmadığı sonucuna varılmıştır [7].

VR-Link katmanı, HLA mimarisinin yanı sıra dağıtık simülasyon sistemlerinin geliştirilmesinde başka bir mimari olan DIS üzerinde de çalışabilen bir katman olarak tasarlanmıştır. Bu mimari bağımsızlık, uygulama geliştiricisine sunulan arayüz (interface) sınıfları ile uygulama katmanından da yalıtılmış durumdadır. Böylece bu katman üzerinde geliştirilen bir simülasyon bileşeninin bu iki mimari arasında geçişi mümkün olmaktadır. Katmanın aynı zamanda FOM bağımsız geliştirilmesi, uygulama kodunun FOM değişikliklerinden minimum düzeyde etkilenmesini sağlamıştır. Bu iki önemli özellik bu çalışmanın diğer iki çalışma üzerindeki önemli avantajları arasında sayılabilir. Diğer taraftan katmanın ticari bir ürün olarak sunulması ve geliştirilen federe başına lisans ücretinin uygulanması özellikle çok bileşenli sistemlerde maliyeti oldukça arttıran bir unsur olmaktadır.

1.4.4 Federe Yönetim Altyapısı

Federe Yönetim Altyapısı (FYA) [8], TÜBİTAK Marmara Araştırma Merkezi Bilişim Teknolojileri tarafından HLA uyumlu simülasyon bileşeni geliştirme altyapısı olarak sunulmuş bir çalışmadır. Bu çalışmada bileşenin RTI bağlantısını sağlayacak bir bağlantı katmanının yanında, yine nesne modelinden otomatik kod üretme teknikleri kullanılarak üretilen bir veri iletişim katmanı ve uygulama kodunun ve olay çevrimlerinin yönetildiği bir üst katman bulunmaktadır. Mimarinin genel görünümü ve katmanlar arasındaki etkileşim Şekil 1.4'de belirtilmiştir.



Şekil 1.4 : Federe Yönetim Altyapısı.

Bu mimari, HLA uyumlu geliştirilebilecek tüm bileşenlerin hem olay temelli hem de zaman yönetimi kavramları kullanılarak geliştirilebilmesine olanak verecek şekilde tasarlanmıştır. Özellikle yönetim katmanı, kullanıcı tarafından geliştirilen ve uygulama katmanında belirli görevleri gerçekleştiren yazılım parçalarının simülasyon katmanı tarafından yönetilebilmesi için bileşen mimarisi içermektedir. Bu mimariye göre uygulama katmanında yer alan ve yönetim katmanında bulunan ana sınıftan türetilen bileşenler simülasyon yöneticisi tarafından simülasyon çevrimine bağlı olarak yönetilebilmektedir. Ancak özellikle yönetim katmanında sunulan bu bileşen hizmeti, HLA zaman yönetim servisinin bu katman tarafından desteklenmesi katmanın performansını olumsuz yönde etkilemektedir. Aynı zamanda her simülasyonda geçerli olmayan senaryo yönetim işlemleri gibi ihtiyaca yönelik hizmetlerin sunulması da bu katmanın genel amaçlı kullanımını olumsuz etkilemektedir.

Federe Yönetim Altyapısı içerisinde yer alan veri iletişim katmanı, bu çalışmada değinilen otomatik kod üretme teknikleri ile benzer şekilde üretilmiş bir katmandır. Ancak bu katmanın nesne modeline bağlı üretilmesi ve FOM değişiklikleri durumunda bu değişiklikleri uygulama katmanından yalıtacak bir çözüm getirmemesi bu katmanın eksikliği olarak gösterilebilir. Katmanda kullanılan kod üretim tekniklerinin veri tipi yerine nesne sınıflarının özelliklerine ve etkileşim sınıflarının parametrelerine bağlı olarak üretilir olması da aynı veri tipine sahip parametre ve özellikler için üretilen kodun defalarca yinelenmesi açısından etkin değildir.

1.5 Hipotez

Oluşturulan altyapının, şablon temelli kod üretme teknikleri ile üretilecek bir katman içermesi ve tüm FEDEP adımları yerine yalnızca HLA OMT temelli olarak sınıfları içerecek bir kütüphane olarak tasarlanması, üretilen katmanın hem daha kolay değiştirilebilmesini, hem de daha kolay kullanılabilmesini sağlayacaktır.

Tüm FEDEP adımlarını içeren ya da HLA tarafından sağlanan zaman yönetimi ve veri dağıtım servisi gibi ileri servisleri destekleyen çalışmalarda, geliştirilecek olan simülasyonun ağ performansı açısından daha az etkin olacağı düşünülmektedir. Bu sınıf kütüphanesi üzerinde ihtiyaç duyulan servisler ilave edilebileceği gibi, ticari ürünlerde bulunan ya da benzer çalışmalarda geliştirilen diğer katmanlarla birlikte ve uyumlu çalışabilecektir.

Altyapının IEEE HLA 1516 standartları ile uyumlu olması nedeniyle, bu altyapı üzerinde geliştirilen uygulamaların mevcut uygulamalarla birlikte çalışabilmesi ve tüm süreci kapsayan ya da HLA servislerini sunan katman yazılımlarından kullanım açısından daha etkin olması beklenmektedir.

2. DAĞITIK SİMÜLASYON SİSTEM MİMARİLERİ VE HLA NESNE MODEL ŞABLONU (OMT)

2.1 Amaç

HLA nesne modellerinin standartlaştırılmış bir şablon yapısında sunulmasının temel nedenleri şu şekilde sıralanabilir:

- İletişimde kullanılan veri ve federeler arasındaki etkileşim hakkında genel bir bilgi verir.
- Federasyona dahil olabilecek federelerin yeteneklerini tanımlamak için standart bir mekanizma sunmaktadır.
- HLA nesne modellerinin tasarlanıp üretilmesinde kullanılabilecek araçlar için tanımlamalar içerir.

HLA nesne modelleri, hem federasyonu oluşturacak federeler için bir simülasyon nesne modelinin (SOM), hem de belirli bir grup federenin bir araya gelmesiyle oluşturulmuş federasyonun tamamı için bir federasyon nesne modelinin (FOM) oluşturulmasında kullanılabilirler. Her iki durumda da nesne model şablonunun amacı simülasyon bileşenlerinin birlikte çalışabilirliğinin ve tekrar kullanılabilirliğinin arttırılmasıdır.

2.2 Federasyon Nesne Modelleri (FOM)

Federasyon tasarım sürecinde, federasyona dahil olan tüm bileşenlerin ihtiyaç duyduğu iletişim gereksinimlerinin belirlenmesi ve anlaşılması önem arz eder. Federasyon nesne modelinin temel amacı, federeler arasında meydana gelen veri alışverişinin standart bir şekilde tanımlanmasını sağlamaktır. Bu verinin içeriği, federasyonda yer alacak tüm nesne (object class) ve etkileşim (interaction class) sınıfları ile bu sınıfları oluşturacak üye değişkenler (attribute) ve parametrelerden (parameter) oluşmaktadır. Bu şekilde, federelerin birlikte çalışabilirliğinin sağlanması amacıyla gerek ancak yeter olmayan bilgi modeli anlaşması (information model contract) sağlanmış olur.

2.3 Simülasyon Nesne Modelleri (SOM)

Federasyon tasarım sürecinde, ihtiyaçların uygun olarak karşılanabilmesi için federasyona dahil olacak her bir federenin yeteneklerinin tanımlanması gerekir. Simülasyon nesne modeli, bir federenin federasyona sağlayabileceği ve federasyondan gereksinim duyacağı verileri belirtmektedir. Bu nesne modellerinin oluşturulması modelin ait olduğu federenin, bir federasyonda görev alıp almayacağının tespit edilmesinde katkı sağlar.

IEEE tarafından tanımlanmış HLA nesne modeli şablonu hem FOM hem de SOM üretilmesinde kullanılabilecek genel bir şablondur. Bu nedenle SOM verilerinin içeriği de federenin ihtiyaç duyduğu ya da sağlayabileceği tüm nesne ve etkileşim sınıfları ile bu sınıfları oluşturacak üye değişkenler ve parametrelerden oluşmaktadır. Bu özellik, bir federasyona katılacak tüm federelerin SOM'larının federasyona ait FOM'un üretilebilmesi amacıyla kullanılabilmesini sağlamaktadır.

2.4 HLA ve Nesne Yönelimli Kavramların İlişkileri

HLA nesne modeli şablonunda yer alan kavramlar nesne yönelimli analiz ve tasarım tekniklerinde var olan nesne modellerine tam anlamıyla karşılık düşmez. Nesne yönelimli alanda, nesne modelleri bir sistemin daha iyi anlaşılması amacıyla kullanılan soyutlama olarak tanımlanır. Bu amaçla, birçok nesne yönelimli teknikte incelenen sisteme farklı bir kaç noktadan yaklaşılması gerekebilir. Diğer taraftan HLA nesne modelleri federasyon içindeki federelere ait veri iletişim istek ve kabiliyetlerini belirtmesi açısından daha dar bir alanı temsil ederler. Bir federe için SOM, federenin sistemin tümüne sunduğu bir arayüz olarak değerlendirilebilir. Bu bilgiler, federenin davranışı ve işleyişi hakkında detay içermezler. Bu tip bilgilerin SOM dışında ek kaynaklar tarafından sunulması gerekir.

Nesne yönelimli alanda, nesneler veri ve metotların birlikte yer aldığı yazılım bileşenleri olarak tanımlanırlar. HLA açısından ise, nesneler federeler arasında gönderilen verilerin yer aldığı bileşenler olarak tanımlanırlar. Bu verileri değiştiren her türlü metot ve davranış federelerin iç işleyişlerinde gerçekleştirilir. Bu nedenle HLA sınıflarında yer alan her türlü üye değişken ve parametre nesne yönelimli programlamada sınıfların üye değişkenlerine karşılık gelmektedir.

HLA sınıflarının koşum anında oluşturulan her örneği (instance), HLA servisleri tarafından bu örneğin ait olduğu nesne/etkileşim sınıfının şablon olarak kullanılmasıyla oluşturulur.

Nesne yönelimli programlamada, nesneler arasındaki etkileşim nesnelerin diğer nesneler üzerinden, sağlanan metotları çağırması ile gerçekleşirken HLA'de etkileşim nesneler arasında değil federeler arasında gerçekleşmektedir. Federeler bu etkileşimi nesne güncellemeleri yaparak ve etkileşim göndererek sağlamaktadır. HLA'de ayrıca bir nesne sınıfına ait örneğin (object instance) üye değişkenlerinin güncellenmesi işlemi federasyonda bulunan farklı bir kaç federe arasında paylaştırılabilmektedir. Nesne yönelimli alanda ise bir nesnenin üye değişkenleri yerel olarak saklanmakta ve üye değişkenlerinin güncellenmesi işlemi bu nesne tarafından sağlanan metotlar yardımıyla nesne tarafından gerçekleştirilmektedir.

2.5 HLA OMT Bileşenleri

HLA nesne modelleri, nesne sınıfları ve üye değişkenleri ile etkileşim sınıfları ve parametrelerini tanımlayan ilişkisel bir kaç bileşenden meydana gelmektedir. Bu bilgi, çeşitli şekillerde sunulabilse de temel olarak şu bileşenlerden oluşmaktadır.

- Nesne Modeli Tanımlama Tablosu (Object Model Identification Table)
- Nesne Sınıfı Yapı Tablosu (Object Class Structure Table)
- Etkileşim Sınıfı Yapı Tablosu (Interaction Class Structure Table)
- Üye Değişken Tablosu (Attribute Table)
- Parametre Tablosu (Parameter Table)
- Boyut Tablosu (Dimension Table)
- Zaman Temsil Tablosu (Time Representation Table)
- Kullanıcı Tanımlı Etiket Tablosu (User-supplied Tag Table)
- Senkronizasyon Tablosu (Synchronization Table)
- İletim Tipi Tablosu (Transportation Type Table)
- Anahtar Tablosu (Switches Table)
- Veri tipi Tablosu (Datatype Table)

- Not Tablosu (Notes Table)
- FOM/SOM Sözlüğü (FOM/SOM Lexicon)

Yukarıda bahsedilen bileşenler, federeler ve federasyonlar için kullanılacak tabloları belirtmektedir. Ancak bu bileşenlerden bazılarının kullanımı her zaman gerekli olmayabilir. Örneğin herhangi bir etkileşim sınıfı ile ilgili olmayan bir federe için SOM dosyasında Etkileşim Sınıfı Yapı Tablosu ve Parametre Tablosu boş olabilir.

2.5.1 Nesne modeli tanımlama tablosu

Nesne modelinin başka modellerin türetilmesinde kullanılabilmesinin sağlanması için bu modele ait ayırt edici bir kaç bilginin modelde var olması gerekir. Yapısı, alanları ve kullanım amaçları Çizelge 2.1'de anlatılmaktadır.

Çizelge 2.1 : Nesne modeli tanımlama tablosu.

Kategori	**Bilgi**	**Amaç**
İsim	<name>	Nesne Modeline verilen ismi belirtir.
Tip	<type>	Nesne modelinin tip bilgisini içerir. (FOM|SOM)
Versiyon	<version>	Versiyon bilgisidir.
Değiştirilme Tarihi	<date>	En son oluşturma ya da değiştirilme tarihini belirtir. (YYYY-MM-DD)
Amaç	<purpose>	Hangi federe/federasyon için hazırlandığı bilgisidir.
Uygulama Alanı	<application domain>	Federenin/Federasyonun hangi amaç için geliştirildiğini belirtir.
Sponsor	<sponsor>	Geliştirmeye sponsor olan organizasyon bilgisidir.
İletişim Noktası	<poc>	Nesne modeli ile ilgili iletişim kurulabilecek kişi bilgisidir.
Organizasyon	<poc organization>	İletişim personeli için kurum bilgisidir.
Telefon	<poc telephone>	İletişim personeli için telefon bilgisidir.
Email	<poc email>	İletişim personeli için e-posta bilgisidir.
Referanslar	<references>	Ek diğer kaynaklara referanslar içeren bölümdür.
Diğer	<other>	Nesne modeline ait diğer açıklayıcı bilgileri içerir.

2.5.2 Nesne sınıfı yapı tablosu

Simülasyon ya da federasyon içindeki nesnelere ait ilişkisel yapıları içeren tablo olarak kullanılır. Federasyonda kullanılacak her bir nesne sınıfını tanımlar. Koşum anında, tanımlanmış olan bu nesne sınıflarının örnekleri (instance) kullanılır. Nesne sınıflarının ilişkileri hiyerarşik tanımlandığı için bu ilişkiler ilgili sütunlarda belirtilir. Ardışık olmayan seviyelerdeki ilişkiler geçişlilik (transivity) özelliği kullanılarak çıkarılabilir.

Nesne sınıfları arasındaki türetme ilişkisi nesne yönelimli yaklaşımda olduğu gibidir. Türetilen bir sınıf ana sınıfının tüm üye değişkenlerini içermekle birlikte ek bazı üye değişkenler de içerebilir. Kendisinden türetilen bir sınıf olmayan tüm nesne sınıfları hiyerarşide yaprak (leaf) sınıf olarak adlandırılır ve nesne modelindeki tüm nesne sınıfları "HLAobjectRoot" sınıfından türemektedirler. Modelde sadece tekil türeme söz konusudur.

Federasyonda bulunan federeler, nesne sınıflarının sınıf hiyerarşisindeki istedikleri özelliklerine abone olabilmektedirler. Nesne modellerindeki hiyerarşik yapı federelere aynı zamanda ilgi duydukları nesne sınıflarına belirli bir seviyede abone olma imkanı da tanımaktadır. Tablonun şablonu Çizelge 2.2'de verilmiştir.

Çizelge 2.2 : Nesne sınıfı yapı tablosu.

HLAobjectRoot (<p/s>)	[<class> (<p/s>)]	[<class> (<p/s>)]	....	[<class> (<p/s>)]
		[<class> (<p/s>)]	....	[<class> (<p/s>)]
		....	....	[<class> (<p/s>)]
	[<class> (<p/s>)]	[<class> (<p/s>)]	....	[<class> (<p/s>)]
	....		....	[<class> (<p/s>)]

Tabloda yer alan sınıfların her biri yayın ya da abone bilgisini belirten parametreler de içerir. SOM için bu parametrelerin alabileceği geçerli değerler şu şekildedir:

- P (Publish): Federe bu sınıfa ait üye değişkenlerin en az birini yayınlayabilir.
- S (Subscribe): Federe bu sınıfa ait üye değişkenlerin en az birine abone olabilir.
- PS (PublishSubscribe): Federe bu sınıfa ait üye değişkenlerin en az birini yayınlayabilir ve en az birine abone olabilir.

- N (Neither): Federe bu sınıfa ait üye değişkenlerin hiç birini yayınlamaz ya da hiç birine abone olmaz.

FOM için aynı parametrelerin geçerli değerleri ise, federasyonda en az bir federenin bu sınıfa ait en az bir üye değişken için yayınlama/abone olma yeteneğine bağlı olarak belirlenir.

2.5.3 Etkileşim sınıfı yapı tablosu

Belirli bir federe tarafından gönderilen ve başka bir federe üzerinde etkiye sahip olabilecek olaylar etkileşim olarak tanımlanır. Bu etkileşimler, nesne sınıflarında olduğu gibi türeme ilişkilerine uygun olarak hiyerarşik bir yapıda ifade edilirler.

Etkileşim sınıfları arasındaki türetme ilişkisi nesne yönelimli yaklaşımda olduğu gibidir. Türetilen bir sınıf ana sınıfının tüm parametrelerini içermekle birlikte ek bazı parametreler de içerebilir. Nesne modelindeki tüm etkileşim sınıfları "HLAinteractionRoot" sınıfından türemektedirler. Tablonun şablonu Çizelge 2.3'de verilmiştir.

Çizelge 2.3 : Etkileşim sınıfı yapı tablosu.

HLAinteractionRoot (<p/s>)	[<class> (<p/s>)]	[<class> (<p/s>)]	....	[<class> (<p/s>)]
		[<class> (<p/s>)]	....	[<class> (<p/s>)]
		....	....	[<class> (<p/s>)]
	[<class> (<p/s>)]	[<class> (<p/s>)]	....	[<class> (<p/s>)]
	....		....	[<class> (<p/s>)]

Etkileşim sınıfları nesne sınıflarından farklı olarak parametre temelli değil sınıf temelli yayınlanıp abone olunurlar. Tabloda yer alan sınıfların her biri yayın ya da abone bilgisini belirten parametreler de içerir. SOM için bu parametrelerin alabileceği geçerli değerler şu şekildedir:

- P (Publish): Federe bu etkileşim sınıfını yayınlayabilir.
- S (Subscribe): Federe bu etkileşim sınıfına abone olabilir.
- PS (PublishSubscribe): Federe bu etkileşim sınıfını yayınlayabilir ve sınıfa abone olabilir.

- N (Neither): Federe bu etkileşim sınıfını yayınlamaz ya da sınıfa abone olmaz.

FOM için aynı parametrelerin geçerli değerleri ise, federasyonda en az bir federenin bu sınıfı yayınlama/abone olma yeteneğine bağlı olarak belirlenir.

2.5.4 Üye değişken tablosu

Simülasyonda yer alan her bir nesne sınıfı belirli bir grup üye değişken ile temsil edilmektedir. Bu değişkenler durum bilgisinin, bu nesnelerin örnekleri üzerinden yayınlanmasını ve RTI tarafından diğer federelere iletilmesini sağlarlar. Nesne sınıfları hiyerarşik yapıda belirtildiğinden üye değişkenlerin bu sınıflara eklenmesinde bu yapı göz önünde tutulmalıdır. "HLAobjectRoot" tüm nesne sınıflarının atası olduğu için üye değişken eklenmesi bu sınıfa da uygulanabilmektedir. Bu üye değişkenler hakkında belirli özelliklerin bilinmesi federeler arasındaki etkileşimin etkinliği açısından önemlidir. Her ne kadar bu değişkenleri tanımlayan veri tipi ve değişkenin güncelleme kuralları RTI tarafından direkt kullanılmasa da, bu bilgiler federelerin birlikte çalışabilirliğinin sağlanması için önemlidir. Tablonun şablonu Çizelge 2.4'de verilmiştir.

Çizelge 2.4 : Üye değişken tablosu.

Nesne	**Üye Değişken**	**Veri Tipi**	**Güncelleme Tipi**	**Güncelleme Koşulu**	**D/A**	**P/S**	**Mevcut Boyutlar**	**İletim**	**Sıra**
HLAobjectRoot	HLAprivilegeToDeleteObject	<datatype>	<update type>	<update condition>	<d/a>	<p/s>	<dimensions>	<transport>	<order>
<object class>	<attribute>	<datatype>	<update type>	<update condition>	<d/a>	<p/s>	<dimensions>	<transport>	<order>
<object class>	<attribute>	<datatype>	<update type>	<update condition>	<d/a>	<p/s>	<dimensions>	<transport>	<order>

Tabloda yer alan ilk sütun nesne sınıfının, nesne sınıfı yapı tablosundaki ismini belirtmektedir. Bu isim nesne sınıfının tekil olarak tanımlanabilmesi için tüm hiyerarşik yapısı içerilecek şekilde kullanılır. Belirli bir üye değişken, karmaşıklığın azaltılması için hiyerarşiye dahil olduğu ilk sınıfın üye değişkeni olarak gösterilir.

İkinci sütun nesne sınıfına ait üye değişkenin belirtildiği sütundur.

Üçüncü sütun üye değişkenin tipini belirten sütundur. Bu veri tipleri, veri tipi tablosunda yer alan basit (simple), numaralandırılmış (enumerated), dizi (array), sabit kayıt (fixed record) veya değişken kayıt (variant record) tipinde olabilir. Güncelleme esnasında değer almayacak değişkenler için bu sütun "NA" değerini alabilir. Bu sütun "NA" değerini aldığında aynı üye değişken için güncelleme tipi, güncelleme koşulu ve mevcut boyutlar sütunları da "NA" değerini alır. Diğer taraftan her üye değişken mutlaka geçerli bir iletim ve sıra değerine sahip olmalıdır.

Dördüncü sütun üye değişkenin güncelleme tipini belirtmektedir ve listelenen değerleri alabilir:

- Statik (Static): Federe bu değeri sadece ilklediğinde ve istenildiğinde günceller.
- Periyodik (Periodic): Düzenli zaman aralıklarında güncellenen değerlerdir.
- Koşullu (Conditional): Belirli koşullar sağlandığında üye değişken güncellemesi yapılır.
- NA: Üye değişkenin güncellenmeyeceği durumlarda kullanılır.

Beşinci sütun üye değişkenin güncelleme koşulunu belirtmektedir. Güncelleme tipinin periyodik olduğu durumlarda, bu sütun birim zamanda yapılacak güncelleme sayısını belirtir. Koşullu güncelleme tipine sahip üye değişkenler için ise bu sütun güncelleme koşulunu içerir. Bir federe, birim zamanda yapacağı güncelleme sıklığını ya da güncelleme koşulunu değiştirebilir nitelikte ise bu durum not tablosunda belirtilir. Güncelleme tipinin statik ya da "NA" olduğu durumlarda güncelleme koşulu sütunu da "NA" değerini alır.

Altıncı sütun üye değişkenin sahipliğinin devredilip devredilemeyeceğini belirtmektedir. FOM için bir üye değişkenin sahipliğinin devredilebilmesi bu üye değişkenin sahipliğinin başka bir federe tarafından alınabilmesini gerektirir. FOM için bu sütunun alabileceği değerler:

- N (NoTransfer): Federasyonda bu üye değişkenin sahipliği devredilemez.
- DA (DivestAcquire): Federasyonda bulunan federelerden bazıları nesne sınıfı örnekleri için bu üye değişkenin sahipliğini devredebilirken bazı federeler de değişkenin sahipliğini alabilir.

SOM için ise bir federe, bir üye değişkenin sahipliğini alabilir, devredebilir, hem devredip hem alabilir ya da bu işlemlerin hiç birini gerçekleştirmez. SOM için bu alanın uygulanabilir değerleri:

- D (Divest): Federe bu üye değişkeni yayınlayıp ilgili HLA servislerini kullanarak sahipliğini başka bir federeye devredebilir.
- A (Acquire): Federe bu üye değişkeni yayınlayıp ilgili HLA servislerini kullanarak sahipliğini başka bir federeden devralabilir.
- N (NoTransfer): Federe ne bu üye değişkenin sahipliğini devralabilir ne de devredebilir.
- DA (DivestAcquire): Federe hem bu üye değişkenin sahipliğini devralabilir hem de devredebilir.

Yedinci sütun, federe ya da federasyon düzeyinde üye değişkenin yayınlanma/abone olunma özelliğini belirtir. SOM için bu sütunun alabileceği değerler:

- P (Publish): Federe bu üye değişkeni yayınlayabilir.
- S (Subscribe): Federe bu üye değişkene abone olabilir.
- PS (PublishSubscribe): Federe bu üye değişkeni yayınlayabilir ve değişkene abone olabilir.
- N (Neither): Federe bu değişkeni yayınlamaz ya da değişkene abone olmaz.

FOM için de parametrelerin aynı değerleri geçerlidir.

Sekizinci sütun üye değişkeni, federe ya da federasyonda veri dağıtım servisleri kullanıldığı durumda, bir grup boyutla ilişkilendirir. Böyle bir durumda bu sütun boyut tablosundaki satırları değer olarak içerir. Bu servisin kullanılmadığı durumda ise “NA” değerini alır.

Dokuzuncu sütun üye değişkenle ilgili kullanılacak iletim tipini belirler ve iletim tablosundan değerler alır.

Onuncu sütun üye değişkenin dağıtımı ile ilgili sıralamayı belirler. Alabileceği değerler:

- Receive: Üye değişken güncellemeleri alıcı federeye belirsiz bir sırada teslim edilir.
- TimeStamp: Üye değişken güncellemeleri alıcı federeye, güncelleme sırasında eklenen zaman pulu (time stamp) bilgisine göre sıralı teslim edilir.

2.5.5 Parametre tablosu

"HLAinteractionRoot" tüm etkileşim sınıflarının atası olduğu için parametre eklenmesi bu sınıfa da uygulanabilmektedir. Ancak nesne sınıflarının üye değişkenlerinden farklı olarak, etkileşim parametreleri, parametre temelli (kısmi) üye olunamaz ve yayınlanamazlar. Bu nedenle etkileşimlere ait boyut, iletim ve dağıtım sırası parametre düzeyinde değil etkileşim sınıfı düzeyinde belirtilmektedir. Çizelge 2.5'de tablonun şablonu verilmiştir.

Çizelge 2.5 : Parametre tablosu.

Etkileşim	**Parametre**	**Veri Tipi**	**Mevcut Boyutlar**	**İletim**	**Sıra**
<interaction class>	<parameter>	<datatype>	<dimensions>	<transport>	<order>
	...	...			
	<parameter>	<datatype>			
<interaction class>	<parameter >	<datatype>	<dimensions>	<transport>	<order>

Tabloda yer alan ilk sütun etkileşim sınıfının, etkileşim sınıfı yapı tablosundaki ismini belirtmektedir. Bu isim, etkileşim sınıfının tekil olarak tanımlanabilmesi için tüm hiyerarşik yapısı içerilecek şekilde kullanılır. Belirli bir parametre, karmaşıklığın azaltılması için hiyerarşiye dahil olduğu ilk sınıfın parametresi olarak gösterilir.

İkinci sütun etkileşim sınıfına ait parametrenin isminin belirtildiği sütundur. Etkileşim sınıfı parametre içermiyorsa "NA" değerini alır.

Üçüncü sütun parametrenin tipini belirten sütundur. Bu veri tipleri, veri tipi tablosunda yer alan basit (simple), numaralandırılmış (enumerated), dizi (array), sabit kayıt (fixed record) veya değişken kayıt (variant record) tipinde olabilir. Etkileşim sınıfı parametre içermiyorsa "NA" değerini alır.

Dördüncü sütun etkileşim sınıfını, federe ya da federasyonda veri dağıtım servisleri kullanıldığı durumda, bir grup boyutla ilişkilendirir. Böyle bir durumda bu sütun boyut tablosundaki satırları değer olarak içerir. Bu servisin kullanılmadığı durumda ise "NA" değerini alır.

Beşinci sütun etkileşim sınıfı ile ilgili kullanılacak iletim tipini belirler ve iletim tablosundan değerler alır.

Altıncı sütun etkileşimin dağıtımı ile ilgili sıralamaya belirler. Alabileceği değerler:

- Receive: Etkileşim alıcı federeye belirsiz bir sırada teslim edilir.
- TimeStamp: Etkileşim alıcı federeye, gönderilme sırasında eklenen zaman pulu (time stamp) bilgisine göre sıralı teslim edilir.

2.5.6 Boyut tablosu

HLA, federeler arasındaki iletişimin etkin olabilmesi için bildirim yönetim servisleri (declaration management) sağlamaktadır. Bu servislerle, federeler sadece ilgili oldukları nesne sınıflarına üye değişken düzeyinde ya da etkileşim sınıflarına abone olabilirler. Ancak çok federeli sistemlerde bu servisler federasyonun veri iletim ihtiyaçlarını karşılamada yetersiz kalabilir. Bu gibi durumlarda veri hacminin azaltılması ve ağ performansının daha etkin kullanılabilmesi HLA tarafından sağlanan veri dağıtım yönetim servisleri kullanılarak (data distribution management) sağlanabilir.

Veri dağıtım yönetim servisinin kullanıldığı her üye değişken ya da etkileşim sınıfı için belirtilen mevcut boyutlar, boyut tablosunda bulunan değerlerin bir alt kümesidir. Bu üye değişkenleri ya da etkileşim sınıfını yayınlayan federeler yayınlamak istedikleri verinin geçerli bölgesini bu boyutlardan oluşan çok boyutlu koordinat sistemini kullanarak tanımlarlar. Benzer şekilde bu üye değişkenlere ya da etkileşim sınıfına abone olan federeler almak istedikleri verinin geçerli bölgesini tanımlamakta da bu boyutları kullanırlar. Bu iki kavram şu şekilde belirtilir:

- Abone Alanları (Subscription Regions): Abone olan federenin ilgi alanını daraltmak için mevcut boyutlar üzerinde tanımlanmış aralık (range) kümelerinden oluşur.

- Güncelleme Alanları (Update Regions): Yayınlayan federenin yayınlama alanını daraltmak için mevcut boyutlar üzerinde tanımlanmış aralık (range) kümelerinden oluşur.

Diğer taraftan, tanımlanan boyutların federeler ve RTI tarafından yorumlanma şekilleri farklıdır. Federeler tanımlanan boyutlar için verilerin temsil edileceği bir veri tipi belirtirler. Bu görüş (view) her federe için aynıdır. Ancak RTI tarafından bu boyut için tanımlanan değerler pozitif tamsayılardan oluştuğu için federeler bu iki görüş arasındaki dönüşümlerin yapılabilmesi için bir normalizasyon fonksiyonu da sağlamalıdırlar. Bu şekilde RTI, boyutlar için federeler tarafından tanımlanmış değerlerden bağımsız olarak kendi görüşü içinde abone alanlarının ve güncelleme alanlarının kesişip kesişmediğini etkin bir şekilde hesaplayabilir. Tablonun şablonu Çizelge 2.6'da verilmiştir.

Çizelge 2.6 : Boyut tablosu.

İsim	**Veri tipi**	**Boyut Üst Limiti**	**Normalizasyon Fonksiyonu**	**Belirtilmemiş Durumlar için Değer**
<dimension>	<type>	<bound>	<normalization function>	<default range/excluded>
<dimension>	<type>	<bound>	<normalization function>	<default range/excluded>

Tabloda yer alan ilk sütun boyutun ismini belirtmektedir. Her boyut, belirli bir değerin filtrelenmesinde kullanılan bir kriteri belirtmektedir.

İkinci sütun boyutun federe görüşünden veri tipini belirtmektedir. Bu veri tipleri veri tipi tablosunda yer alan basit (simple), numaralandırılmış (enumerated), dizi (array), sabit kayıt (fixed record) veya değişken kayıt (variant record) tipinde olabilir.

Üçüncü sütun boyut için ihtiyaç duyulan ve RTI tarafından kullanılacak olan üst limiti belirtmektedir.

Dördüncü sütun abone/yayın koordinatlarının alt ve üst limitlerinden, pozitif tamsayılardan oluşan [0, boyut üst limiti) aralığına eşleştirme metodunu tanımlamaktadır. Her boyut için bu normalizasyon metodunun, federasyona katılan tüm federeler tarafından aynı şekilde kullanılması önemlidir.

Beşinci sütun boyut için bir değer belirtilmediği durumlarda RTI tarafından bölge kesişimleri için kullanılacak varsayılan değer aralığını belirtir. Burada belirtilen aralık alt sınırı kapsayacak ancak üst sınırı dışlayacak şekilde belirtilir. Bu alana yazılan "Excluded" değeri, boyut için bir değer belirtilmediği durumlarda bu alanın da kesişim hesaplarında kullanılmayacağını belirtmektedir.

2.5.7 Zaman temsil tablosu

Simülasyonlar bir sistemin zaman içindeki davranışının modellenmesi amacıyla kullanılırlar. Bu nedenle sistemde kullanılacak zamanın temsil edilmesi önem taşımaktadır. Federeler kendilerini HLA'in zaman ekseni ile ilişkilendirip belirli olayları bu eksendeki belirli noktalarla eşleyebilirler ve federasyon işletimi boyunca bu eksen üzerinde zamanlarını ilerletebilirler.

Simülasyonlarda kullanılan zaman yönetimi stratejisi, birlikte çalışabilirlik açısından önem taşımaktadır. Bazı simülasyonlar gerçek zamanlı işletilirken bazıları, olay temelli ya da gerçek zamandan daha hızlı (ölçeklendirilmiş) işletilebilmektedirler. Bu nedenle bir simülasyonun zaman temsilinin nasıl yapılacağının SOM'da belirtilmesi önemlidir.

HLA tarafından sağlanan zaman yönetimi servisi, federelerin iç işleyişlerinde zaman ilerletme stratejileri nasıl olursa olsun bu federelerin birlikte çalışabilmelerine olanak verecek şekilde tasarlanmıştır. Bu amaçla federasyon genelinde kullanılacak zaman pullarının (time stamp) nasıl temsil edileceğinin FOM'da belirtilmesi gerekir.

Zaman temsil tablosu, RTI tarafından federeler/federasyon içinde kullanılacak zaman pulunun nasıl kullanılacağını ve hangi veri tipi ile temsil edileceğini belirtmektedir. Bunun yanı sıra yine RTI tarafından federelerin ve federasyonların zaman ilerletme işlemlerinde kullanılacak olan lookahead değerinin de hangi veri tipi ile temsil edileceği ve ne şekilde kullanılacağı bu tabloda temsil edilmektedir. Tablo detayları Çizelge 2.7'de verilmiştir.

Çizelge 2.7 : Zaman temsil tablosu.

Kategori	**Veri tipi**	**Semantik**
Time stamp	<type>	<semantics>
Lookahead	<type>	<semantics>

Tabloda yer alan ilk sütun zamanla ilgili iki kavramın (time stamp, lookahead) kategorisinin belirtildiği sütundur.

İkinci sütun bu kavramların temsil edilecekleri veri tipini belirtmektedir. Bu veri tipleri, veri tipi tablosunda yer alan basit (simple), numaralandırılmış (enumerated), dizi (array), sabit kayıt (fixed record) veya değişken kayıt (variant record) tipinde olabilir. Federe ya da federasyon tarafından bu kavramın uygulanabilir olmaması durumunda "NA" değerini alır.

Üçüncü sütun kavrama ait veri tipinin kullanımı ile ilgili semantik bilgileri içerir. Federe ya da federasyon tarafından bu kavramın uygulanabilir olmaması durumunda bu sütun "NA" değerini alır.

2.5.8 Kullanıcı tanımlı etiket tablosu

RTI, HLA tarafından belirtilen servislerin kullanımında daha ayrıntılı kontrol ve koordinasyonun sağlanması amacıyla ek bilgiler sağlanmasına olanak verir. Bu tablo, temel olarak bu ek bilgilerin sağlanması amacıyla kullanılır. Tablo detayları Çizelge 2.8'de verilmiştir.

Çizelge 2.8 : Kullanıcı tanımlı etiket tablosu.

Kategori	**Veri tipi**	**Semantik**
Update/reflect	<type>	<semantics>
Send/receive	<type>	<semantics>
Delete/remove	<type>	<semantics>
Divesture request	<type>	<semantics>
Divesture completion	<type>	<semantics>
Acquisition request	<type>	<semantics>
Request update	<type>	<semantics>

İlk sütun kullanıcı tarafından ek bilgi sağlanabilecek HLA servislerinin kategori bilgisini içerir.

- Update/Reflect: Nesne sınıfı örneklerinin üye değişkenlerinin güncellenmesi ve alınması işlemidir.
- Send/Receive: Etkileşim sınıflarının gönderilip alınması işlemidir.
- Delete/Remove: Nesne sınıfı örneklerinin silinip kaldırılması işlemidir.
- Divesture Request: Nesne sınıfı örneklerinin üye değişkenlerinin sahipliğinin devir isteğidir.
- Divesture Completion: Üye değişkenlerin devir işlemlerinin tamamlanmasıdır.

- Acquisition Request: Nesne sınıfı örneklerinin üye değişkenlerinin sahipliğinin alınma isteğidir.
- Request Update: Nesne sınıfı örneklerinin üye değişkenlerinin güncellenme isteğidir.

İkinci sütun bu etiketlerin temsil edilecekleri veri tipini belirtmektedir. Bu veri tipleri, veri tipi tablosunda yer alan basit (simple), numaralandırılmış (enumerated), dizi (array), sabit kayıt (fixed record) veya değişken kayıt (variant record) tipinde olabilir. Federe ya da federasyon tarafından bu kavramın uygulanabilir olmaması durumunda bu sütun "NA" değerini alır.

Üçüncü sütun etikete ait veri tipinin kullanımı ile ilgili semantik bilgileri içerir. Federe ya da federasyon tarafından bu kavramın uygulanabilir olmaması durumunda "NA" değerini alır.

2.5.9 Senkronizasyon tablosu

RTI federeler arasında belirli etkinliklerin senkronize edilmesi amacıyla servisler sunmaktadır. Senkronizasyonun sağlanmasında kullanılacak olan senkronizasyon noktalarının ve bunlar hakkındaki bilgilerin tanımlanmasında senkronizasyon tablosu kullanılmaktadır. Tablonun detayları Çizelge 2.9'da verilmiştir.

Çizelge 2.9 : Senkronizasyon tablosu.

Etiket	**Veri tipi**	**Yetkinlik**	**Semantik**
<label>	<type>	<capability>	<semantics>
<label>	<type>	<capability>	<semantics>

İlk sütun senkronizasyon noktasına ait isim bilgisini içermektedir.

İkinci sütun senkronizasyon noktası kullanılırken federe ya da federasyon tarafından sağlanacak kullanıcı etiketine ait veri tipini belirtmektedir. Bu veri tipleri, veri tipi tablosunda yer alan basit (simple), numaralandırılmış (enumerated), dizi (array), sabit kayıt (fixed record) veya değişken kayıt (variant record) tipinde olabilir.

Üçüncü sütun, federenin senkronizasyon noktasında gösterebileceği yetkinlikleri tanımlamaktadır ve federasyon için uygulanmaz. Bu nedenle FOM'da "NA" değerini taşır. SOM için alabileceği değerler:

- Register: Federe senkronizasyon noktası kayıt etme yeteneğine sahiptir.
- Achieve: Federe kayıtlı senkronizasyon noktasına ulaşma yeteneğine sahiptir.

- Register Achieve: Federe hem senkronizasyon noktası kaydı hem de kayıtlı noktaya ulaşma yeteneğine sahiptir.
- NoSynch: Federe ne senkronizasyon noktası kaydı ne de kayıtlı noktaya ulaşma yeteneğine sahiptir.

Dördüncü sütun noktaya ve kullanımına ait daha detaylı bilgilerin sunulmasını sağlar.

2.5.10 İletim tablosu

RTI, veri iletimi için farklı mekanizmalar sunmaktadır. Bu tablo RTI tarafından verilerin iletimi esnasında kullanılacak olan bu mekanizmaların tanımlanmasında kullanılır. IEEE Std 1516.1-2000 standardı, HLA tarafından kullanılacak olan iki mekanizmayı (HLAreliable, HLAbestEffort) tanımlamaktadır. Diğer iletim tipleri bu tablo yardımı ile eklenebilmektedir. Tablo detayları Çizelge 2.10'da verilmiştir.

Çizelge 2.10 : İletim tablosu.

İsim	**Tanım**
HLAreliable	Veri iletimi TCP/IP kullanılarak sağlanmaktadır.
HLAbestEffort	Veri iletimi UDP kullanılarak sağlanmaktadır.
<name>	<desciption>

İlk sütun veri iletim mekanizmasının ismini belirtmektedir.

İkinci sütun, veri iletim yöntemine ait tanımlama bilgisini içermektedir.

2.5.11 Anahtar tablosu

RTI federeler tarafından gerçekleştirilecek bazı eylemlerin başlayıp durdurulması amacıyla hizmetler de sunar. Bu hizmetler, federe ve federasyon istekleri doğrultusunda açılıp kapatılabilmektedir. Bunlar arasında, yeni bir nesne sınıfı örneği keşfedildiğinde güncellemelerin otomatik olarak istenmesi (Auto Provide anahtarı tarafından kontrol edilir), ve belirli olaylara göre federelere bildirim yapılması (Advisory anahtarı tarafından kontrol edilir) sayılabilir. Anahtar tablosu, bu anahtarların ilk durumlarının belirlenmesini sağlar ve anahtar değerleri koşum sırasında değiştirilebilirler.

Anahtar değerlerinin çoğu, federelere özgü olup Auto Provide ve Convey Region Designator Sets anahtarları federasyona özgüdür ve bir federe tarafından değerinin değiştirilmesi federasyonun geneline etki etmektedir. Tablo detayları Çizelge 2.11'de verilmiştir.

Çizelge 2.11 : Anahtar tablosu.

Anahtar	**Ayar**
Auto Provide	<auto provide>
Convey Region Designator Sets	<convey region designator sets>
Attribute Scope Advisory	<attribute scope advisory>
Attribute Relevance Advisory	<attribute relevance advisory>
Object Class Relevance Advisory	<object class relevance advisory>
Interaction Relevance Advisory	<interaction relevance advisory>
Service Reporting	<service reporting>

İlk sütun, değerleri değiştirilebilecek anahtarların isimlerini içermektedir. Bu anahtarların kullanım amaçları şu şekilde özetlenebilir:

- Auto Provide: RTI tarafından yeni bir nesne sınıfı örneği keşfedildiğinde, bu örneğin üye değişkenlerinin sahibinden otomatik güncellemeyle istenip istenmeyeceğini belirtir.
- Convey Region Designator Sets: RTI tarafından güncelleme yansıtmalarında (Reflect Attribute Values) ve etkileşim alımlarında (Receive Interaction) opsiyonel olan "Sent Region Set" parametresinin gönderilip gönderilmeyeceğini belirler.
- Attribute Scope Advisory: Nesne örneğinin üye değişkenlerinin kapsam giriş ya da çıkışlarında RTI tarafından federelere bildirim yapılıp yapılmayacağını belirler.
- Attribute Relevance Advisory: Nesne örneğinin belirli bir üye değişkeni için güncelleme sağlayıp sağlamayacağının RTI tarafından bildirilmesidir. RTI, bu kararı üye değişkene başka federeler tarafından ihtiyaç duyulup duyulmamasına bağlı olarak verir.
- Object Class Relevance Advisory: Federeler tarafından nesne sınıflarının örneklerinin kayıt edilip edilmeyeceğinin bildirilmesidir. Bu karar, nesne sınıfının üye değişkenlerinden herhangi birine başka federeler tarafından ihtiyaç duyulup duyulmamasına bağlı olarak verilir.

- Interaction Relevance Advisory: Federeler tarafından etkileşim sınıflarının örneklerinin gönderilip gönderilmeyeceğinin bildirilmesidir. Bu karar, etkileşim sınıfına başka federeler tarafından ihtiyaç duyulup duyulmamasına bağlı olarak verilir.
- Service Reporting: Yönetim nesne modeli (Management Object Model-MOM) kullanılarak RTI tarafından servislerin raporlanıp raporlanmayacağını belirten anahtardır.

İkinci sütun anahtarlar için ayar değerlerini belirtir. SOM için uygulanamayan durumlar için "NA" değerini alırken, FOM için tüm değerler listelenen geçerli değerler arasından seçilmelidir.

- Enabled: Anahtar etkin durumdadır. RTI tarafından, uygun durumlarda kullanılacaktır.
- Disabled: Anahtar pasif durumdadır. RTI tarafından kullanılmayacaktır.

2.5.12 Veri tipi tablosu

Birçok OMT tablosunda (üye değişken, parametre, boyut, zaman temsil, kullanıcı tanımlı etiket ve senkronizasyon) veri tipi sütunu bulunmaktadır. Veri tipi tablosu, ilgili tablolarda kullanılan veri tiplerini tanımlayan bilgileri içermektedir. Bu tablolarda yer alan her bir veri tipi, basit veri tipi (simple datatype table), numaralandırılmış veri tipi (enumerated datatype table), dizi veri tipi (array datatype table), sabit kayıt veri tipi (fixed record datatype table) ya da değişken kayıt veri tipi (variant record datatype table) tablolarının herhangi birinde bir kayda karşılık gelmektedir. Sayılan bu tablolar daha karışık veri tiplerinin oluşturulabilmesi amacıyla diğer tablolarda yer alan veri tiplerini de içerebilmektedirler.

2.5.12.1 Temel veri temsil tablosu

Tüm veri tipleri için temel sayılabilecek verilerden oluşmaktadır. Tanımlanan her FOM ve SOM'da nesne modeli tarafından kullanılmasa da bulunmaktadır. Çizelge 2.12'deki ön tanımlı değerleri içermekle birlikte yeni tipler tanımlanabilmektedir:

Çizelge 2.12 : Temel veri temsil tablosu.

İsim	**Boyut (Bit)**	**Yorumlama**	**Endian**	**Kodlama**
HLAinteger16BE	16	[-2^{15}, 2^{15}-1] aralığında tamsayı	Big	16-bit ikiye tümleyen işaretli tamsayı. En yüksek anlamlı bit işareti taşımaktadır.
HLAinteger32BE	32	[-2^{31}, 2^{31}-1] aralığında tamsayı	Big	32-bit ikiye tümleyen işaretli tamsayı. En yüksek anlamlı bit işareti taşımaktadır.
HLAinteger64BE	64	[-2^{63}, 2^{63}-1] aralığında tamsayı	Big	64-bit ikiye tümleyen işaretli tamsayı. En yüksek anlamlı bit işareti taşımaktadır.
HLAfloat32BE	32	Tek-hassas kayan noktalı sayı	Big	32-bit IEEE normalizasyonlu tek-hassas format (IEEE Std. 754-1985).
HLAfloat64BE	64	Çift-hassas kayan noktalı sayı	Big	64-bit IEEE normalizasyonlu tek-hassas format (IEEE Std. 754-1985).
HLAoctetPairBE	16	16-bit değer	Big	Tüm donanımlara uyumlu olacağı varsayılmaktadır.
HLAinteger16LE	16	[-2^{15}, 2^{15}-1] aralığında tamsayı	Little	16-bit ikiye tümleyen işaretli tamsayı. En yüksek anlamlı bit işareti taşımaktadır.
HLAinteger32LE	32	[-2^{31}, 2^{31}-1] aralığında tamsayı	Little	32-bit ikiye tümleyen işaretli tamsayı. En yüksek anlamlı bit işareti taşımaktadır.
HLAinteger64LE	64	[-2^{63}, 2^{63}-1] aralığında tamsayı	Little	64-bit ikiye tümleyen işaretli tamsayı. En yüksek anlamlı bit işareti taşımaktadır.
HLAfloat32LE	32	Tek-hassas kayan noktalı sayı	Little	32-bit IEEE normalizasyonlu tek-hassas format (IEEE Std. 754-1985).
HLAfloat64LE	64	Çift-hassas kayan noktalı sayı	Little	64-bit IEEE normalizasyonlu tek-hassas format (IEEE s 754-1985).
HLAoctetPairLE	16	16-bit değer	Little	Tüm donanımlara uyumlu olacağı varsayılmaktadır.
HLAoctet	8	8-bit değer	Big	Tüm donanımlara uyumlu olacağı varsayılmaktadır.
<name>	<size>	<interpretation>	<endian>	<encoding>
<name>	<size>	<interpretation>	<endian>	<encoding>

İlk sütun temel veri tipinin ismini belirtmektedir.

İkinci sütun, veri tipinin bit olarak boyutunu belirtmektedir.

Üçüncü sütun, veri tipinin yorumlanış şeklini belirtmektedir.

Dördüncü sütun, veri tipinin bayt dizilimini belirtmektedir. Listelenen değerler geçerlidir.

- Big: Yüksek anlamlı bayt ilk gelir.
- Little: Düşük anlamlı bayt ilk gelir.

Beşinci sütun, RTI gönderim ve alımı esnasında verinin bit diziliminin detaylarını belirtmektedir.

2.5.12.2 Basit veri tipi tablosu

Basit veri tiplerinin tanımlamalarından oluşmaktadır. Tanımlanan her FOM ve SOM'da bulunmaktadır. Çizelge 2.13'deki ön tanımlı değerleri içermekle birlikte yeni tipler tanımlanabilmektedir:

Çizelge 2.13 : Basit veri tipi tablosu.

İsim	**Temsil**	**Birim**	**Çözünürlük**	**Doğruluk**	**Semantik**
HLAASCIIchar	HLAoctet	NA	NA	NA	
HLAunicodeChar	HLAoctetPairBE	NA	NA	NA	
HLAbyte	HLAoctet	NA	NA	NA	
<simple type>	<representation>	<units>	<resolution>	<accuracy>	<semantics>
<simple type>	<representation>	<units>	<resolution>	<accuracy>	<semantics>

İlk sütun basit veri tipinin ismini belirtmektedir.

İkinci sütun, veri tipinin temel veri tiplerinden hangisi ile temsil edilebileceğini belirtmektedir. Temel veri temsili tablosundan değerler alır.

Üçüncü sütun, kullanılacak veri tipinin birimini belirtmektedir. Bu birim çözünürlük ve doğruluk sütunları için de geçerlidir. Bu alanın uygulanabilir olmaması durumunda "NA" değeri yazılır.

Dördüncü sütun, veri tipinin ayrık iki değeri arasında olabilecek minimum farkı belirtir. Uygulanamayan veri tipleri için "NA" değerini alır.

Beşinci sütun, RTI gönderim ve alımı esnasında verinin esas değerinden sapabileceği maksimum değeri gösterir. Ayrık veri tipleri için "perfect" değerini alırken uygulanamayan veri tipleri için "NA" değerini alır.

Altıncı sütun veri tipinin kullanım detayları ile ilgili bilgiler içerir. Açıklama gereği olmayan veri tipleri için "NA" değerini alır.

2.5.12.3 Numaralandırılmış veri tipi tablosu

Numaralandırılmış veri tipleri sonlu ayrık değer kümelerinden değer alabilecek değişkenler için kullanılırlar. Tanımlanan her FOM ve SOM'da nesne modeli tarafından kullanılmasa da bulunmaktadır. Çizelge 2.14'deki ön tanımlı değeri (HLAboolean) içermekle birlikte yeni tipler tanımlanabilmektedir:

Çizelge 2.14 : Numaralandırılmış veri tipi tablosu.

İsim	**Temsil**	**Numaralayıcı**	**Değerler**	**Semantik**
HLAboolean	HLAinteger32BE	HLAfalse	0	Standart bool tipi
		HLAtrue	1	
<enumerated type>	<representation>	<enumerator 1>	<value(s)>	<semantics>
		…	…	
		<enumerator n>	<value(s)>	

İlk sütun numaralandırılmış veri tipinin ismini belirtmektedir.

İkinci sütun, veri tipinin temel veri tiplerinden hangisi ile temsil edilebileceğini belirtmektedir. Temel veri temsili tablosundan değerler alır.

Üçüncü sütun, veri tipi ile ilişkilendirilmiş tüm numaralayıcıların isimlerini içerir.

Dördüncü sütun, her numaralayıcı için karşı gelen değeri içerir. Değerler ikinci sütunda belirtilen temsil ile uyumlu olmalıdır.

Beşinci sütun, veri tipinin kullanım detayları ile ilgili bilgiler içerir. Açıklama gereği olmayan veri tipleri için "NA" değerini alır.

2.5.12.4 Dizi veri tipi tablosu

Aynı tipe sahip verilerin tutulduğu dizileri belirten tablodur. Tanımlanan her FOM ve SOM'da nesne modeli tarafından kullanılmasa da bulunmaktadır. Çizelge 2.15'deki ön tanımlı değerleri içermekle birlikte yeni tipler tanımlanabilmektedir:

Çizelge 2.15 : Dizi veri tipi tablosu.

İsim	Eleman tipi	Eleman sayısı	Kodlama	Semantik
HLAASCIIstring	HLAASCIIchar	Dinamik	HLAvariableArray	ASCII katar temsili
HLAunicodeString	HLAunicodeChar	Dinamik	HLAvariableArray	Unicode katar temsili
HLAopaqueData	HLAbyte	Dinamik	HLAvariableArray	Yorumlanmamış bayt sekansı

İlk sütun dizi veri tipinin ismini belirtmektedir.

İkinci sütun, dizi elemanlarının veri tipinin temel veri tiplerinden hangisi ile temsil edilebileceğini belirtmektedir. Bu veri tipi basit (simple), numaralandırılmış (enumerated), dizi (array), sabit kayıt (fixed record) veya değişken kayıt (variant record) tipinde olabilir.

Üçüncü sütun, dizinin eleman sayısıdır. Eleman sayıları sabit diziler için bu değer kullanılırken, eleman sayısı değişkenlik gösteren dizilerde "Dynamic" değerini alır.

Dördüncü sütun, dizi veri tipinin gönderilmesi ve alınması esnasında kullanılacak olan kodlama detaylarını belirtmektedir. Dizideki elemanların kodlama işlemi elemanın veri tipine uygun gerçekleştirilir. Dizinin kodlama işlemi ise bu sütunda yer alan mekanizmaya uygun gerçekleştirilir. Tek boyutlu dizilerde bu sütun "HLAfixedArray" ve "HLAvariableArray" olmak üzere iki adet öntanımlı değer alabilir. "HLAfixedArray", sabit eleman sayılı diziler için kullanılır ve elemanların dizideki sıralarına uygun olarak kodlanmasını öngörür. "HLAvariableArray", değişken eleman sayılı diziler için tanımlanır. Dizinin eleman sayısının HLAinteger32BE olarak kodlanmasından sonra, dizi elemanlarının dizideki sıralarına uygun olarak kodlanmasını esas alır.

Beşinci sütun, dizi veri tipinin kullanım detayları ile ilgili bilgiler içerir. Açıklama gereği olmayan veri tipleri için "NA" değerini alır.

2.5.12.5 Sabit kayıt veri tipi tablosu

Bu tabloda yer alan veri tipleri farklı tipteki verilerin oluşturduğu yapılardır. Bu yapıdaki her veri tipi basit (simple), numaralandırılmış (enumerated), dizi (array), sabit kayıt (fixed record) veya değişken kayıt (variant record) tipinde olabilir. Bu durum, kullanıcılara yapılardan oluşan yapılar (structures of data structures) tanımlama imkanı vermektedir. Detayları Çizelge 2.16'da verilmiştir.

Çizelge 2.16 : Sabit kayıt veri tipi tablosu.

Kayıt İsmi	**Alan**			**Kodlama**	**Semantik**
	İsim	**Tip**	**Semantik**		
<record type>	<field 1>	<type 1>	<semantics>	<encoding>	<semantics>
	...	...	...		
	<field n>	<type n>	<semantics>		
<record type>	<field 1>	<type 1>	<semantics>	<encoding>	<semantics>
	...	...	...		
	<field m>	<type m>	<semantics>		

İlk sütun sabit kayıt veri tipinin ismini belirtmektedir.

İkinci sütun, kayıttaki alanın veri ismini belirtmektedir.

Üçüncü sütun, kayıttaki alanın veri tipini belirtmektedir. Bu veri tipi basit (simple), numaralandırılmış (enumerated), dizi (array), sabit kayıt (fixed record) veya değişken kayıt (variant record) tipinde olabilir.

Dördüncü sütun, alanın kullanımı ile ilgili bilgileri içerir. Bu bilginin olmadığı durumlarda "NA" değerini alır.

Beşinci sütun, sabit kayıtlı veri tipinin gönderilmesi ve alınması esnasında kullanılacak olan kodlama detaylarını belirtmektedir. Bu kayıttaki alanların kodlama işlemi, alan için belirtilen veri tipine uygun olarak gerçekleştirilir. Sabit kayıt için kullanılacak kodlama mekanizması kodlama sütununda belirtildiği şekilde uygulanır. Daha sonra alanların veri tiplerine uygun olan kodlama işlemleri gerçekleştirilir. Varsayılan kodlama mekanizmasını kullanan sabit kayıtlar için bu sütun "HLAfixedRecord" değerini alır. Bu değer, sabit kayıtta yer alan her alanın tanımlandığı sırada kodlanmasını belirtmektedir.

Altıncı sütun, veri tipinin kullanım detayları ile ilgili bilgiler içerir. Açıklama gereği olmayan veri tipleri için "NA" değerini alır.

2.5.12.6 Değişken kayıt veri tipi tablosu

Çizelge 2.17'de detayları verilen bu kayıt tipi, veri tiplerinin ayrık birleşimlerini (union) temsil etmek amacıyla kullanılmaktadır.

Çizelge 2.17 : Değişken kayıt veri tipi tablosu.

Kayıt İsmi	**Ayırt Edici Özellik (Discriminant)**			**Alternatif**			**Kodlama**	**Semantik**
	İsim	**Tip**	**Numara layıcı**	**İsim**	**Tip**	**Semantik**		
<variant type>	<name>	<type>	<set 1>	<name 1>	<type 1>	<semantics>	<encoding>	<semantics>
			...	...	...	...		
			<set n>	<name n>	<type n>	<semantics>		
<variant type>	<name>	<type>	<set 1>	<name 1>	<type 1>	<semantics>	<encoding>	<semantics>
			...	...	...	...		
			<set m>	<name m>	<type m>	<semantics>		

İlk sütun değişken kayıt veri tipinin ismini belirtmektedir.

İkinci sütun, ayırt edici özelliğin ismini belirtmektedir.

Üçüncü sütun, ayırt edici özelliğin veri tipini belirtmektedir. Bu veri tipi numaralandırılmış (enumerated) veri tipinde olabilir.

Dördüncü sütun, numaralayıcı için değer kümesini gösterir. Bu değerler üçüncü sütunda belirtilen numaralayıcı tipinde olmalıdırlar.

Beşinci sütun, alternatif için isim değerini taşımaktadır. Herhangi bir alternatifin bulunmaması durumunda, "NA" değerini almaktadır.

Altıncı sütun alternatif için veri tipini belirtir. Bu veri tipleri, veri tipi tablosunda yer alan basit (simple), numaralandırılmış (enumerated), dizi (array), sabit kayıt (fixed record) veya değişken kayıt (variant record) tipinde olabilir. Herhangi bir alternatifin bulunmaması durumunda bu sütun "NA" değerini alır.

Yedinci sütun alternatifin kullanımı ile ilgili semantik bilgileri içerir. Bu kavramın uygulanabilir olmaması durumunda bu sütun "NA" değerini alır.

Sekizinci sütun, değişken kayıtlı veri tipinin gönderilmesi ve alınması esnasında kullanılacak olan kodlama detaylarını belirtmektedir. Alternatiflerin kodlama işlemi, alan için belirtilen veri tipine uygun gerçekleştirilir. Değişken kayıt için kullanılacak kodlama mekanizması kodlama sütununda belirtildiği şekilde uygulanır. Daha sonra alanların veri tiplerine uygun olan kodlama işlemleri gerçekleştirilir. Varsayılan kodlama mekanizmasını kullanan kayıtlar için bu sütun "HLAvariantRecord" değerini alır. Bu değer, kayıtta yer alan her ayırt edici özelliğin, bu özelliğin değeri ile ilişkilendirilmiş olan alternatif tarafından takip edildiğini belirtmektedir.

Son sütun, veri tipinin kullanım detayları ile ilgili bilgiler içerir. Açıklama gereği olmayan veri tipleri için "NA" değerini alır.

2.5.12.7 Birleşik veri tipleri için öntanımlı kodlamalar

Her birleşik veri tipi (diziler, sabit kayıt, değişken kayıt) için bu yapıdaki elemanların birbirlerine göre nasıl dizilmesi gerektiğini belirten kodlamalar bulunmaktadır. Bu kodlama mekanizmaları, yapı içerisinde yer alan her bileşenin düzgün bayt yerleşiminin sağlanması için kendinden sonra gelen bileşenden önce nasıl düzenlenmesi (padding) gerektiğini belirtmektedir.

Kodlama mekanizması olarak HLAfixedArray, HLAvariableArray, HLAfixedRecord, HLAvariableRecord seçilmesi bayt dizilimlerinin de bu mekanizmalara uygun gerçekleştirilmesi gerektiğini belirtmektedir.

Genel anlamda, birleşik veri yapıları içerisindeki her bileşenin düzgün diziliminin sağlanması için, ihtiyaç duyulan bileşenlerden sonra ek baytların (padding bytes) eklenmesi gerekir. Örnek olarak 32-bit kayan noktalı sayılar 32 bit sınırlarına göre düzenlenmelilerken, 64-bit kayan noktalı sayılar 64 bit sınırlarına göre düzenlenmelidirler. Bir bileşenin bayt düzenini belirleyen üç unsur bulunmaktadır:

- Bileşenin, birleşik veri tipinin başlangıcından itibaren içerdiği offset bayt sayısı
- Bileşenin bayt olarak boyutu
- Bir sonraki bileşenin "sekizli sınır değeri" (octet boundary value)

Basit ve numaralandırılmış veri tipleri için sekizli sınır değeri, bu veri tipi için temel veri temsil tablosunda karşılık gelen satıra göre belirlenir. Sekizli sınır değeri; n, $8*2^n$ değerini veri tipinin bit olarak boyutundan büyük ya da bu değere eşit yapacak en küçük pozitif tamsayı olmak üzere, 2^n olarak tanımlanır. Öntanımlı temel veri tipleri için sınır değerleri Çizelge 2.18'de verilmiştir.

Çizelge 2.18 : Temel veriler için sekizli sınır değer tablosu.

Temel Temsil	**Sekizli sınır değeri**
HLAoctet	1
HLAoctetPairBE	2
HLAinteger16BE	2
HLAinteger32BE	4
HLAinteger64BE	8
HLAfloat32BE	4
HLAfloat64BE	8
HLAoctetPairLE	2
HLAinteger16LE	2
HLAinteger32LE	4
HLAinteger64LE	8
HLAfloat32LE	4
HLAfloat64LE	8

Birleşik veri tipleri için sınır değeri, bu yapıda bulunan tüm bileşenler içinde sınır değeri en büyük olana göre belirlenir. Bu değer, birleşik veri tipi için de sınır değeri olarak tanımlanır. Örnek olarak, HLAboolean (HLAinteger32BE olarak temsil edilen), HLAoctet ve HLAfloat64BE veri tiplerinden oluşan bileşenler içeren birleşik veri tipi için, bu sınır değeri 8 olacaktır.

Sonraki bölümlerde, öntanımlı her birleşik veri tipi için kodlama mekanizmaları ve kullanılması gereken dizilim baytlarını içeren hususlar ele alınmıştır. Bit olarak boyutu, 8'in katı olmayan veri tipleri ile kullanılması durumunda düzenleme bitleri (padding bits) kullanılarak boyut 8'in katı olacak şekilde düzenlenir. Hem düzenleme bitleri hem de düzenleme bayları 0 değerleri ile temsil edilirler.

HLAfixedRecord: Bu kodlama mekanizması her alanın tanımlandığı sırada kodlanması gerektiğini belirtmektedir. Kayıttaki ilk alan için offset değeri 0 olacaktır.

Son alan dışındaki her alana, sonraki alanın düzgün dizilimi için sıfır ya da daha fazla dizilim baytı eklenir. Sabit kayıttaki i. alan için aşağıdaki formülü sağlayan en küçük P_i değeri, eklenmesi gereken dizilim baytlarının sayısını verir.

$$(Offset_i + Size_i + P_i) \bmod V_{i+1} = 0 \qquad \textbf{(2.1)}$$

Formülde yer alan değerler aşağıda tanımlanmıştır:

$Offset_i$: Sabit kaydın i. alanı için offset değeridir (bayt olarak).

$Size_i$: i. alanın boyutudur (bayt olarak).

V_{i+1} : (i+1). alan için sekizli sınır değeridir.

Son alan dizilim baytlarını içermez ve sabit kayıt boyutu alanlardan sonra eklenen dizilim baytlarını da kapsar.

Sırasıyla HLAoctet, HLAboolean ve HLAfloat64BE veri tiplerinden oluşan alanlar içeren ve HLAfixedRecord kodlama mekanizmasını kullanan sabit kayıt için bayt dizilim görünümü Şekil 2.1'de verilmiştir:

Bayt															
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
HLAoctet	0	0	0	HLAboolean				HLAfloat64BE							

Şekil 2.1 : Sabit kayıt bayt dizilim görünümü.

HLAvariantRecord: Bu kodlama mekanizması, değişken kayıtta yer alan her ayırt edici özelliğin, bu özelliğin değeri ile ilişkilendirilmiş olan alternatif tarafından takip edildiğini belirtmektedir. Ayırt edici özellik için offset değeri 0 olacaktır.

Ayırt edici özellik için alternatiflerin bulunmadığı durumlarda, özellikten sonra gerekli durumlarda dizilim baytları eklenir. Alternatiflerin bulunduğu durumlarda ise, yine gerekli durumlarda, takip eden alternatifin düzgün dizilimi için dizilim baytları eklenir. Bu baytların sayısı aşağıdaki formülü sağlayan en küçük pozitif P değeri ile belirlenir.

$$(Size + P) \bmod V = 0 \qquad \textbf{(2.2)}$$

Formülde yer alan değerler aşağıda tanımlanmıştır:

Size: Ayırt edici özelliğin boyutudur (bayt olarak).

V: Alternatifler içinde sekizli sınır değerlerinin en büyüğüdür.

Alternatiften sonra dizilim baytları eklenmez ve değişken kayıt boyutu eklenen dizilim baytlarını da kapsar.

Ayırt edici özellik olarak, HLAinteger32BE tipinde alternatifin kullanıldığı değişken kayıt için bayt dizilim görünümü Şekil 2.2'de verilmiştir:

Bayt							
0	1	2	3	4	5	6	7
HLAoctet	0	0	0	HLAinteger32BE			

Şekil 2.2 : Değişken kayıt bayt dizilim görünümü.

HLAfixedArray: Bu kodlama mekanizması, boyutu sabit olan dizilerde, dizideki her elemanın sırasıyla kodlanmasını öngörmektedir. Dizinin ilk elemanı için offset değeri 0 olacaktır.

Son eleman dışındaki her eleman için, sonraki elemanın düzgün diziliminin sağlanması amacıyla sıfır ya da daha fazla dizilim baytı eklenir. Dizi içindeki i. eleman için aşağıdaki formülü sağlayan en küçük P_i değeri, eklenmesi gereken dizilim baytlarının sayısını verir.

$$(Size_i + P_i) \bmod V = 0 \qquad \textbf{(2.3)}$$

Formülde yer alan değerler aşağıda tanımlanmıştır:

$Size_i$: i. elemanın boyutudur (bayt olarak).

V : dizi elemanlarının veri tipi için sekizli sınır değeridir.

Son eleman dizilim baytlarını içermez ve dizinin boyutu elemanlardan sonra eklenen dizilim baytlarını da kapsar.

Elemanları, HLAinteger32BE ve HLAoctet tipinde iki alan içeren sabit kayıtlar olan 2 elemanlı bir dizi için bayt dizilim görünümü Şekil 2.3'de verilmiştir:

Bayt												
0	1	2	3	4	5	6	7	8	9	10	11	12
HLAinteger32BE				HLAoctet	0	0	0	HLAinteger32BE				HLAoctet

Şekil 2.3 : Sabit boyutlu dizi için bayt dizilim görünümü.

HLAvariableArray: Bu mekanizma eleman sayısı değişken olan diziler için tanımlanmıştır. Bu kodlama mekanizmasında, ilk olarak dizideki eleman sayısı HLAinteger32BE tipinde kodlandıktan sonra her eleman veri tipine uygun olarak sırayla kodlanır. Dizinin eleman sayısı için offset değeri 0 olacaktır.

Dizinin eleman sayısından sonra gerekli durumlarda dizilim baytları eklenir. Bu dizilim baytlarının sayısı aşağıdaki formülü sağlayan en küçük P değeri olarak tanımlanır.

$$(4 + P) \bmod V = 0 \qquad \textbf{(2.4)}$$

Formülde yer alan V değeri aşağıda tanımlanmıştır:

V: dizi elemanlarının veri tipi için sekizli sınır değeridir.

Dizinin her elemanından sonra eklenecek dizilim baytlarının sayısı HLAfixedArray mekanizmasındaki gibi hesaplanır ve dizinin boyutu eklenen dizilim baytlarını da kapsar.

Elemanları 24-bit büyüklüğünde alanlardan oluşan değişken uzunluktaki bir dizi için bayt dizilim görünümü Şekil 2.4'de verilmiştir:

Bayt										
0	1	2	3	4	5	6	7	8	9	10
HLAinteger32BE				24-bit değer			0	24-bit değer		

Şekil 2.4 : Değişken boyutlu dizi için bayt dizilim görünümü.

2.5.13 Not tablosu

Nesne model şablonlarında yer alan her tablo satırı için ayrıntılı bilgilerin sağlanması not tablosu ile yapılır. OMT tablolarında referans verilmiş her bir not için, not tablosu bir satır içermektedir. Bu tabloda yer alan bir not birden fazla OMT tablo satırı ile eşleşebileceği gibi, bir OMT satırı için bu tabloda birden çok not da bulunabilmektedir. Detayları Çizelge 2.19'da verilmiştir.

Çizelge 2.19 : Not tablosu.

Etiket	**Semantik**
<label>	<semantics>
<label>	<semantics>
<label>	<semantics>

İlk sütun, OMT tablolarında referans verilen her not için atanan etiketi belirtmektedir.

İkinci sütun, nota ait açıklama bilgisini içermektedir.

2.5.14 FOM/SOM sözlüğü

Birlikte çalışabilirliğin sağlanması kullanılacak veri yapılarının ve tiplerinin tanımlanmasının yanında bu verinin kullanımına yönelik ortak bir anlayışın sağlanmasını da gerektirmektedir. FOM/SOM sözlüğü, nesne sınıfları, etkileşim sınıfları, üye değişkenler ve parametreler hakkında bu ortak anlayışın sağlanmasında kullanılmaktadır.

2.5.14.1 Nesne sınıfı tanımlama tablosu

Nesne sınıflarının tanımlanması için kullanılır. Detayları Çizelge 2.20'de verilmiştir.

Çizelge 2.20 : Nesne sınıfı tanımlama tablosu.

Sınıf	**Tanım**
<class>	<definition>
...	...
<class>	<definition>

İlk sütun FOM/SOM nesne sınıfının ismini belirtmektedir.

İkinci sütun sınıfa ait tanımlama bilgisini içermektedir.

2.5.14.2 Etkileşim sınıfı tanımlama tablosu

Etkileşim sınıflarının tanımlanması için kullanılır. Detayları Çizelge 2.21'de verilmiştir.

Çizelge 2.21 : Etkileşim sınıfı tanımlama tablosu.

Sınıf	**Tanım**
<class>	<definition>
...	...
<class>	<definition>

İlk sütun FOM/SOM etkileşim sınıfının ismini belirtmektedir.

İkinci sütun sınıfa ait tanımlama bilgisini içermektedir.

2.5.14.3 Üye değişken tanımlama tablosu

Nesne sınıflarının üye değişkenlerinin tanımlanması için kullanılır. Detayları Çizelge 2.22'de verilmiştir.

Çizelge 2.22 : Üye değişken tanımlama tablosu.

Sınıf	**Üye Değişken**	**Tanım**
<class>	<attribute>	<definition>
...	...	...
<class>	<attribute>	<definition>

İlk sütun üye değişkenin ait olduğu FOM/SOM nesne sınıfının ismini belirtmektedir.

İkinci sütun üye değişkenin ismini belirtmektedir.

Son sütun üye değişkene ait tanımlama bilgisini içermektedir.

2.5.14.4 Parametre tanımlama tablosu

Etkileşim sınıflarının parametrelerinin tanımlanması için kullanılır. Detayları Çizelge 2.23'de verilmiştir.

Çizelge 2.23 : Parametre tanımlama tablosu.

Sınıf	**Üye Değişken**	**Tanım**
<class>	<parameter>	<definition>
...	...	...
<class>	<parameter>	<definition>

İlk sütun, parametrenin ait olduğu FOM/SOM etkileşim sınıfının ismini belirtmektedir.

İkinci sütun parametrenin ismini belirtmektedir.

Son sütun parametreye ait tanımlama bilgisini içermektedir.

3. KOD ÜRETİMİ

Kod üretme yüksek düzeydeki programlama dillerinde (C, C++, Java gibi) derlenebilecek kaynak kod dosyalarının üretilmesi işlemidir. Özellikle karmaşık ve yoğun programlama gerektiren işlerin kolaylaştırılması, üretilme şekli belirli yapılardan ya da şablonlardan türetilebilen akışların oluşturulması, kullanıcı tarafından hatalı kodlanabilecek düzeyde detay içeren ve birçok kullanıcının uygulama geliştirme esnasında kullanabileceği kaynak kodların ya da yazılım kütüphanelerinin oluşturulmasında yaygın olarak kullanılmaktadır.

Kod üretme teknikleri, ortak erişim ya da iletişim amaçlı kullanılan arakatman yazılımlarının (veritabanı erişim katmanı, dağıtık mimari haberleşme katmanı) geliştirilmesi, tekrar kullanılabilecek kod parçalarının üretilmesi, kullanıcı arayüzlerinin dinamik olarak üretilmesi ve birim test üretilmesi gibi değişik birçok alanda kullanılabilmektedir.

Dağıtık mimarilerde çalışan sistemler için kod üretimi, kullanıcıların farklı seviyelerde ihtiyaç duyacakları iletişim ile ilgili detaylı işlemlerin defalarca yapılması yerine, tüm sistem genelinde belirlenmiş olan iletişim kurallarına uygun olarak kodlanmış ve uygulama kodlarına bağlanabilecek kütüphanelerin oluşturulması amacıyla kullanılmaktadır. Dağıtık sistemlerin heterojen yapısı göz önüne alındığında bu tip yazılımların geliştirilmesinin zor, zaman alıcı ve hataya açık olduğu söylenebilir [9].

3.1 Kod Üretiminin Yazılım Mühendisliği Açısından Önemi

Kod üretim sürecinin yazılım mühendisliği açısından değerlendirilmesi bir kaç farklı seviye için aşağıdaki başlıklarda açıklanabilir:

- Kalite: Büyük ölçekli yazılımlarda, yazılımın kalite açısından aynı seviyede seyretmesi her zaman mümkün olmamaktadır. Bunun temel nedenleri arasında, geliştirme esnasında kalite kriterlerinin sürece her zaman etkin şekilde uygulanamayışı, geliştiricilerin geliştirme esnasında buldukları yeni yaklaşımlar sayılabilir. Diğer taraftan kod üretme işlemlerinde belirli bir şablonun kullanılması, bu şablonun ihtiyaçlar ya da kalite açısından duyulan yeni gereksinimler ya da iyileştirmeler doğrultusunda değiştirilmesi durumunda, değişikliğin yazılımın geneline uygulanabilmesini mümkün kılar.
- Tutarlılık: Kod üretme sonucu oluşturulan kaynak kodlar, özellikle uygulama programlama arayüzlerinde (API) tutarlılığın ve anlaşılabilirliğin sağlanması açısından önemlidir.
- Tekrar Kullanılabilirlik: Genel amaçlı ihtiyaçlar doğrultusunda hazırlanan kod üreticiler vasıtasıyla oluşturulan kaynak kodlar ya da kütüphaneler, tekrar kullanılabilirlik açısından oldukça önemlidir.
- Tek Noktadan Müdahale: Yazılım sistemlerinin zaman içerisinde geliştirilmesi ve değiştirilmesi çok iyi tasarlanmış sistemlerde dahi değişimin başlangıç noktasından itibaren, yazılım içersindeki birçok noktayı etkilemesi ihtimalini doğurur. Bu durum bazen yapılan değişikliğin sistemin çalışabilirliğini olumsuz etkilemesine neden olmaktadır. Ancak, bu değişikliğin kod üretim tekniğinin üzerine kurulduğu şablonlarda yapılabildiği durumlarda değişiklik tek noktadan yapıldığı halde, etkinin yeniden üretilen kodun geneline tutarlı bir şekilde yansıması mümkündür.
- Daha Fazla Tasarım Zamanı: Elle kodlamanın yapıldığı projelerde, proje takvimleri tanımlanan iş için gerekli kod satır sayısı (LOC) ve adam*ay hesabı yapılarak hazırlanır. Bu planlarda özellikle test sürecinin de tekrarlı işlemesinden dolayı sistemin tasarımı ve metotların etkinliği konusundaki çalışmalara çok zaman ayrılamamaktadır. Tüm bu etkiler proje ölçeği büyüdükçe zamandan sapmalarının artmasına ve proje ilerleyişinin aksamasına neden olmaktadır. Kod üretme teknikleri ise, geliştiricilerin sistemin tasarımına ayıracakları zamanı önemli ölçüde arttırmakta ve sistemin kod üretimine uygun olarak tasarlanması durumunda tekrarlı işleyen süreçlerin çok daha hızlı bitirilmesine imkan vermektedir.

- Tasarım Kararlarının Değerlendirilmesi: Uygulama analistlerinin ve kod geliştiricilerinin birlikte yer aldığı sistemlerde, öntasarımda belirlenen kararların kodlama esnasında değiştirilmesinin kaçınılmaz oluşu ve zaman geçtikçe bu değişikliklerin tasarıma yansıtılmasının zorlaşması, analistler ve geliştiriciler arasında ortak tasarım anlayışının yürütülmesini zorlaştırır. Kod üretme sürecinde özellikle tasarımın ortak noktaları ve işleyişleri hakkında üretilebilecek otomatik dökümanlar, tasarım-kod tutarlılığının sağlanması ve analistler ile geliştiriciler arasındaki iletişimin devamı açısından önemlidir.
- Maliyet Etkinliği: Kod üretiminin kullanımı, projelerde çok sık rastlanan satın alma/geliştirme kararlarına da etki etmektedir. Satın alınacak ticari yazılımların lisanslama ücretleri, ihtiyaçlar doğrultusunda değiştirilmesi, kullanımının zaman alıcı olabilmesi dolayısı ile bu kararların iyi değerlendirilmeleri gerekmektedir. Kod üretme proje yönetimini bu açıdan önemli ölçüde etkilemektedir.
- Bağımlılığın Azaltılması: Geliştirilen projelerde kullanılan üçüncü parti yazılımlar, çıkan ürünün bu ürünlere bağımlı olmasına, kullanılan ürünlerdeki versiyon değişiklikleri de bazı durumlarda ürünün de değiştirilmesine neden olmaktadır. Kod üretiminin kullanılması bu bağımlılıkların azaltılmasına katkı sağlayacaktır.
- İdame Kolaylığı: Geliştirilen ürünün başka yazılım kütüphanelerine bağımlılığının azaltılması, ürünün dağıtım ve idame sürecini de önemli ölçüde kolaylaştıracaktır.

3.2 Kod Üretme Teknikleri

Kod üreticiler genel olarak çalışma prensipleri açısından iki kategoriye ayrılırlar.

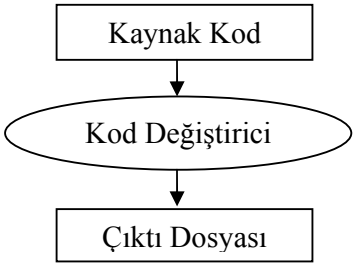
- Pasif Model: Bu modeldeki üreticiler, geliştiricilerin kodlama esnasında değiştirebilecekleri kod üretmektedirler. Bu yolla üretilen kodlar geliştirmede başlangıç noktası olarak kullanılırlar ve kısa ya da uzun dönemde bakım zorunluluğu getirmezler. Entegre Geliştirme Ortamlarında (IDE) sağlanan sihirbazlar bu üreticilere örnek olarak gösterilebilir.

- Aktif Model: Bu modelde üretici, aynı girdi üzerinde defalarca çalıştırılabilmekte ve zaman içinde girdi değiştirilerek kod tekrar üretilebilmektedir. Değişiklikler geliştiriciler tarafından girdilere yansıtılarak üretilen kod güncellenebilmekte ve yeniden kullanılabilmektedir.

İlerleyen bölümlerde aktif model üzerinde geliştirilen kod üreticiler için yaygın kullanılan metotlar ve kullanım amaçları verilmiştir.

3.3 Kod Değiştirme (Code Munging)

Belirli bir giriş kodu için, belirli sayıda ve tipte çıkış üreten üreticilerdir. Akış açısından en temel modeldir. Giriş dosyaları, üretici tarafından düzenli deyimler (regular expression) ya da belirli kod kalıpları açısından değerlendirilir. Çıkış, hazırlanmış olan belirli şablon dosyalarına ya da kodlanmış olan formata uygun olarak üretilir [10]. İşleyişi Şekil 3.1'de verilmiştir.

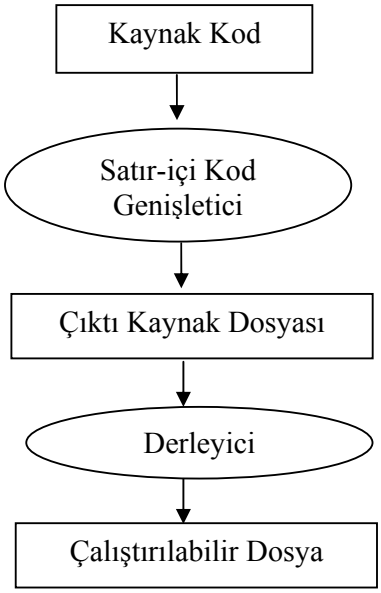


Şekil 3.1 : Kod değiştirme akışı.

Bu üreticiler döküman oluşturma, fonksiyon ya da sabitler için prototip tanımlarının belirli dosyalardan okunması amacıyla kullanılabilmektedir.

3.4 Satır-içi Kod Genişleticiler (Inline-Code Expander)

Kod üretici bu modelde, girdi kaynak dosyalarında bulunan özel bölümleri çıktıda karşı gelecek kod blokları ile değiştirerek çıktı kaynak dosyalarını oluşturur [10]. İşleyişi Şekil 3.2'de verilmiştir.

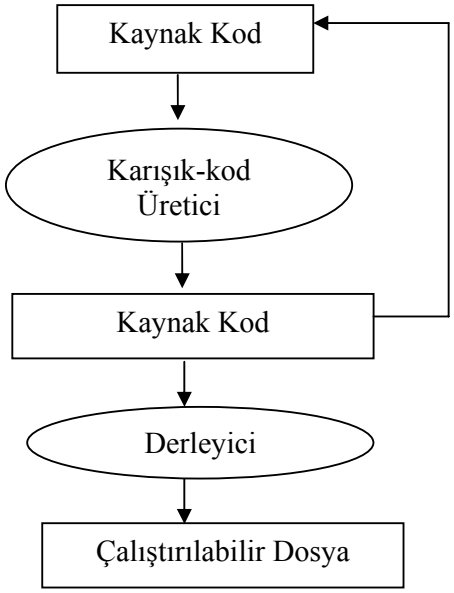


Şekil 3.2 : Satır-içi kod genişletme akışı.

Bu metot, SQL deyimlerinin kaynak kod dosyalarına gömülmesi amacıyla yaygın olarak kullanılmaktadır. Geliştiriciler, çalıştırılmasını istedikleri SQL deyimi ile ilgili, girdi dosyalarında özel işaret/deyimler kullanarak üreticinin bu işaretleri çıktı dosyasında SQL deyimleri ile değiştirmesini sağlayabilirler.

3.5 Karışık-Kod Üretimi (Mixed-Code Generation)

Girdi kaynak dosyalarında gerekli değişiklikler yapılarak çıktı dosyaların üretilmesine dayanır. Kod genişleticilerden farklı olarak çıktı dosyaları süreç içinde girdi olarak tekrar kullanılabilmektedir. Bu tip üreticiler, kaynak koddaki özel alanları çıktı kod dosyası için değiştirerek çıktı dosyalarını üretir [10]. İşleyişi Şekil 3.3'de verilmiştir.

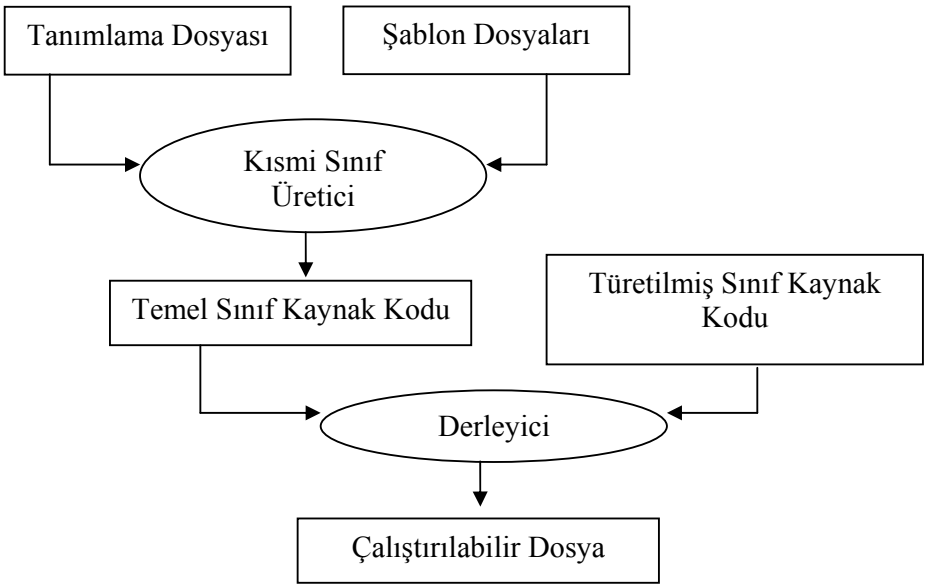


Şekil 3.3 : Karışık-kod üretimi akışı.

Karışık kod üreticiler, genel olarak sıralama (marshalling) kodlarının yazılması amacıyla kullanılırlar. Örneğin arayüz alanlarına karşı gelecek uygulamadaki veri yapılarının doldurulması amacıyla gerekli kodun üretilmesi bu üreticiler vasıtasıyla yapılabilir. Bu amaçla üretici, bu iki yapı arasında gerekli eşlemeleri belirten yorum deyimlerini kullanabilir.

3.6 Kısmi-Sınıf Üretimi (Partial-Class Generation)

Kısmi sınıf üreticileri belirli bir grup sınıfın üretilmesi amacıyla gerekli olan tanımlama (definition) dosyalarını ve çıktı için hazırlanmış olan şablon dosyalarını kullanarak temel sınıfların (base class) bulunacağı kütüphane kodunun oluşturulmasında kullanılırlar. Bu kütüphane daha sonra geliştiriciler tarafından kullanılacak sınıfların türetilmesinde temel olarak kullanılır [10]. İşleyişi Şekil 3.4'de verilmiştir.

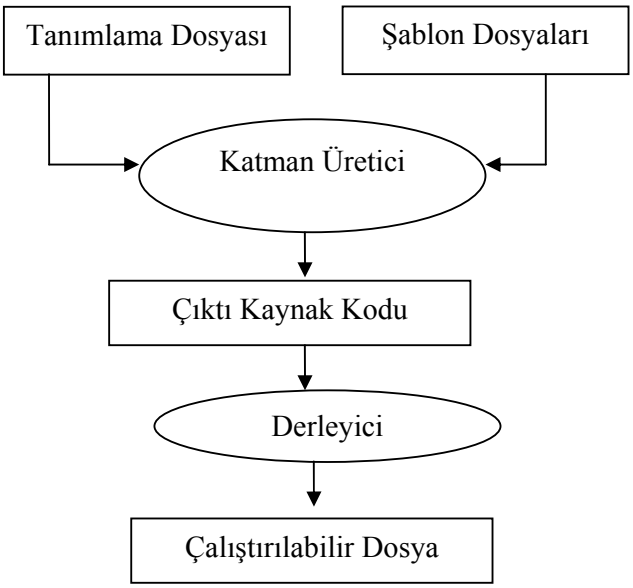


Şekil 3.4 : Kısmi-sınıf üretimi akışı.

Bu model, özellikle katmanlı yapılardan oluşan mimarilerde belirli bir katmandaki temel sınıfların oluşturulması amacıyla kullanılabilir.

3.7 Tabaka ya da Katman Üretimi (Tier or layer generation)

Bu modelde üretici n-katmanlı bir mimari için gerekli bir katmanın tümü ile üretilmesinden sorumludur. Bu model, genellikle model temelli üretim (model-driven generation) amacıyla kullanılmaktadır. Katmanlı mimarinin UML ile hazırlanması, kod üreticinin buna uygun olarak tasarlanması ve gerekli girdi dosyalarının kullanılması adımlarının uygulanmasıyla çıktı dosyalarının üretilmesini sağlar [10]. İşleyişi Şekil 3.5'de verilmiştir.



Şekil 3.5 : Katman üretimi akışı.

Kısmi kod üreteci ile katman üreteci arasındaki en büyük fark, kısmi kod üretiminde belirli bir katmanı oluşturan temel sınıflar üretilip gerekli eklemeler türetilmiş sınıfların kodlanması ile gerçekleştirilirken, katman üretiminde bir katmanın tümü ile üretilmesi kod üreteci tarafından sağlanır. Kısmi kod üretimi genel olarak daha basit bir akış gerektirdiğinden, katman kodu üretimi için bir başlangıç noktası olarak kullanılabilir [10].

3.8 Tam Uygulama Alanı Dili (Full-domain language)

Bu modelle üretilen kodlar Turing dilleri niteliklerini taşır. Bu dillerde, her türlü değişken yönetimi, mantık işlemleri, dallanmalar ve fonksiyonellik bulunmaktadır. Kod üretimi genel olarak yüksek seviyeli dillerde derlenebilir kaynak kodların üretilmesi olarak tanımlansa da Turing dillerinin bu metotla üretilmesi de bu kapsamda değerlendirilebilir. Bu yolla geliştiricilere, uygulama alanındaki kavramların geliştirme dünyasında temsil edilebilmesi imkanı sağlanmış olur. Ancak üretilen kodun anlaşılması ve kullanılabilmesi geliştiriciler açısından zaman alıcı olabilmektedir. Aynı zamanda geliştirilen bu yüksek-seviyeli dil ile mevcut dillerde sağlanan döküman ve test durumu üretimi işlemleri sağlanamayabilir [10].

4. KOD ÜRETİMİNİN NESNE ŞABLONLARINA UYGULANABİLİRLİĞİ

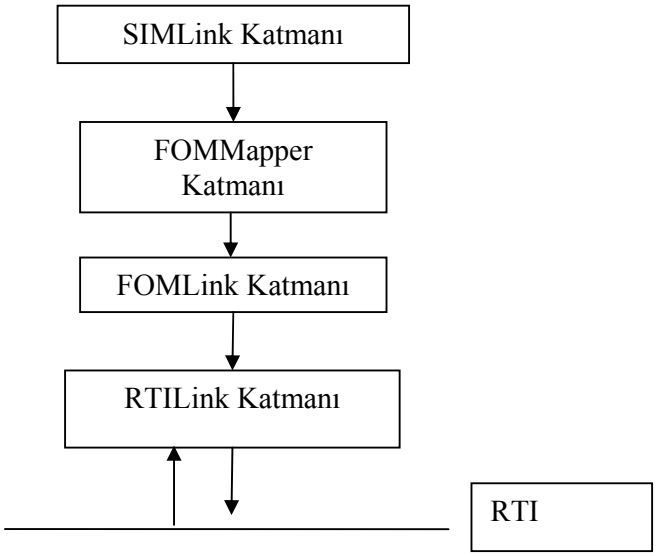
Nesne model şablonu simülasyon sistemleri tarafından nesne ve etkileşim sınıfları ile bu sınıfların üye değişken ve parametrelerinin hiyerarşik bir şekilde tanımlanmasını sağlar. Bu model aynı zamanda, her değişkenin temsil edileceği veri tipinin, değişkenin ağ üzerinde aktarım şeklini ve dağıtım esnasında kullanılması gereken sıralama metotlarını da içerir. Değişkenlerin temsil edildiği tüm veri tipleri için bu verinin ağ üzerinde alınıp verilmesi sırasında ne şekilde kodlanıp çözüleceği de bu şablon tarafından tanımlanmaktadır. Birleşik veri tipleri de temelde kendilerini oluşturan temel veri tiplerinin belirli sırada kodlanıp çözülmesi ile alınıp verilmektedir.

Nesne model şablonunda yer alan sınıfların bu şekilde hiyerarşik tanımlanmaları ve bu sınıfları oluşturan veri tipleri için de kullanılması gereken kodlama ve çözme mekanizması bu şablonda detaylı olarak tanımlandığı için, bu sınıflara karşı gelecek programlama dili sınıflarının üretilmesi ve bu sınıfların üye değişkenlerinin ağ üzerinde gönderilip alınması işlemi otomatik kod üretimi açısından uygun görülmektedir. Aşağıdaki akış bu işlemin gerçekleştirilmesi için gerekli adımları içermektedir:

- Nesne şablonunda yer alan sınıflar için hiyerarşinin oluşturulması.
- Sınıflar içinde kullanılan veri tipleri için hiyerarşinin oluşturulması.
- Oluşturulan veri tipleri için kodlayıcı/çözücü metotlarının kodlanması.
- Üretilen sınıflar için erişim ve değişim metotlarının gerçeklenmesi.

Bu adımda kullanılacak sınıflar ve verilerin kodlama/çözme işlemlerinin gerçekleştirilmesi amacıyla gerekli olacak sınıflar için tanımlamalar şablonlar kullanılarak oluşturulabilir. Bundan sonraki adımda, çıkarılan hiyerarşiye uygun olarak gerekli kodlar üretilerek hazırlanan şablonlara uygulanır ve programlama amacıyla kullanılacak sınıflar elde edilmiş olur. Bu sınıflar, üretilen yardımcı sınıflar vasıtası ile ağ üzerinden gönderilip alınabilmektedir.

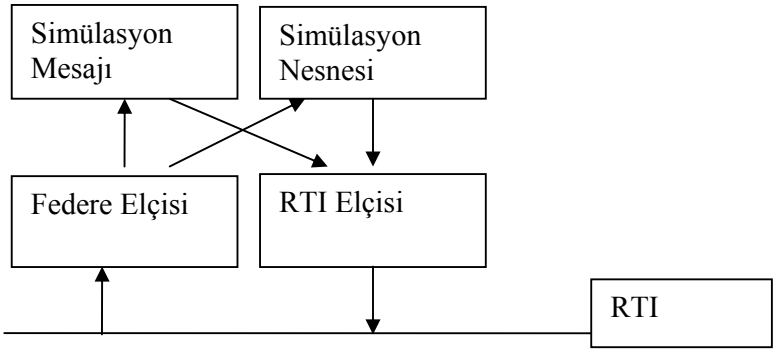
Çalışmanın bundan sonraki bölümünde, kod üretme teknikleri kullanılarak nesne modeline uygun olarak üretilmiş katmanı da içeren Simülasyon Geliştirme Altyapısından bahsedilecek ve altyapının benzer platformlarla kıyaslamasına değinilecektir. Şekil 4.1'de bu altyapıda bulunan katmanlar gösterilmiş olup ilerleyen bölümlerde her bir katman ayrı başlıklarda ele alınarak detaylandırılmıştır.



Şekil 4.1 : Simülasyon Geliştirme Altyapısı.

4.1 RTILink Katmanı

Simülasyon bileşeninin RTI bağlantısını sağlamak, bileşenin RTI üzerinden yapacağı temel çağrılar için arayüz metotları ile bileşen üzerinde RTI tarafından çağrılacak olan arayüz metotlarının gerçekleştirildiği katman olarak tasarlanmıştır. Federenin simülasyon bağlantısını sağlayan en alt seviye katman olarak kullanılmaktadır. Böylece, RTI üzerinden federasyona dahil olabilecek bir federe belirli parametreler sağlanarak gerçekleştirilebilmektedir. Yapısı Şekil 4.2'de gösterilmiştir.



Şekil 4.2 : RTILink katmanı.

RTI Elçisi sınıfı, federenin RTI üzerinde çağırabileceği temel servisleri gerçekleştirmek üzere sağlanan sınıftır. Sınıfın federe tarafından RTI üzerinde yapılacak çağrılar amacıyla sağladığı temel metotlar ve işlevleri aşağıdaki verilmiştir.

- *createFedEx*: Federenin, dahil olacağı federasyonun belirli bir isim ile oluşturulması amacıyla RTI üzerinden yaptığı çağrıdır. Simülasyon sistemlerinde aynı federasyona dahil olacak her bir federe giriş sırasından bağımsız olarak bu çağrıyı yapabilmektedir. Federasyonun oluşturulması bu çağrıyı ilk yapan federe tarafından gerçekleştirilmektedir.
- *joinFedEx*: Oluşturulmuş bir federasyona federenin dahil olabilmesi amacıyla RTI'a federe tarafından yapılan çağrı metodudur. Bu çağrı ile birlikte federe, belirtilen federasyona dahil olur ve RTI tarafından kendisine tekil bir tanımlayıcı atanır.
- *evokeFederateAmbassadorCallbacks:* Federenin RTI üzerinden yaptığı bu çağrı, RTI'a, federe elçisi üzerinden federe tarafında çağırmak istediği metotlar için gerekli zamanın tanınması amacıyla sağlanmıştır. Simülasyon verilerinin federeye yansıtılması ya da federenin RTI'a, RTI elçisi üzerinden yaptığı bir isteğin sonucunun federeye bildirilmesi bu metotla mümkün olmaktadır.

- *reserveName*: Federe tarafından simüle edilecek nesne sınıfı örneklerinin belirli bir isimle RTI'a kayıt ettirilmesi isteği bu metot ile sağlanmaktadır. Kayıt işleminin sonucu federe elçisi üzerinden, başarılı olması durumunda *objectInstanceNameReservationSucceeded* metodu, başarısız olması durumunda ise *objectInstanceNameReservationFailed* metodu çağrılarak bildirilecektir.
- *registerObject*: Federe tarafından simüle edilecek ve RTILink katmanında Simülasyon Nesnesi sınıfı ile temsil edilen bir nesne sınıfı örneğinin simülasyon ortamına kayıt ettirilmesi işleminin yapıldığı metottur.
- *fedTickObject*: Kaydı yapılmış nesne sınıfının kodlanmış özellik güncellemelerinin simülasyon ortamına aktarıldığı metottur. Güncellemesi yapılan her nesne sınıfı RTILink katmanında Simülasyon Nesnesi sınıfı ile temsil edilmektedir.
- *deleteObject:* Federenin simülasyon ortamına yayınlamakta olduğu bir nesne sınıfı örneğinin silinmesi işlemi bu metot üzerinden RTI'a çağrı yapılarak gerçekleştirilmektedir.
- *fedSendMessage:* Federe tarafından, yayınlanabileceği nesne model şablonunda belirtilmiş olan etkileşim sınıflarının simülasyon ortamına kodlanmış olarak gönderilmesi bu metot ile gerçekleştirilmektedir. Her etkileşim sınıfı RTILink katmanındaki Simülasyon Mesajı sınıfı ile temsil edilmektedir.
- *resignAndDestroy:* Federenin federasyondan ayrılma işlemlerinin yapıldığı metotdur. Federasyondan ayrılan her federe, RTI'a federasyonun yok edilmesi amacıyla bir istekte bulunabilir. Federasyonun yok edilmesi işlemi, bu istekte bulunan son federe tarafından gerçekleştirilmektedir.

Federe Elçisi sınıfı, RTI tarafından federe üzerinde çağrılacak olan ve RTI üzerinden sisteme dahil olacak her bileşenin gerçekleştirmesinin zorunlu olduğu metotların sağlandığı sınıftır. Bu metotlar, RTI tarafından sağlanmış soyut (*abstract*) bir arayüz sınıfı içerisinde soyut metotlar (*pure virtual*) olarak tanımlanmış ve simülasyona katılacak her federe tarafından gerçeklenmesi gereken metotlardır. Sınıfın, RTI tarafından federe üzerinde yapılacak çağrılar amacıyla sağladığı temel metotlar ve işlevleri aşağıdaki verilmiştir.

- *objectInstanceNameReservationSucceeded*: Nesne örneklerinin simülasyon ortamında kullanacakları isimlerin tekil olması amacıyla bu isimlerin RTI' kayıt ettirilmesi gerekmektedir. RTI, bu kayıt işleminin sonucunun başarılı olması durumunda federe elçisi üzerinden bu metodu çağıracaktır.
- *objectInstanceNameReservationFailed*: Nesne örneği isim kaydının başarısız olması durumunda RTI, federe elçisi üzerinden bu metodu çağıracaktır.
- *discoverObjectInstance*: Simülasyon ortamında federe tarafından abone olunmuş nesne sınıfı tipinden yeni bir örnek oluşması durumunda federeye bu örneğe ait bilgilerin aktarılacağı keşif metodudur. Federe Elçisi, simülasyondaki her nesne sınıfı örneğini, RTILink katmanında yer alan Simülasyon Nesnesi tipinden bir değişken olarak temsil etmektedir.
- *provideAttributeValueUpdate*: Federenin simülasyona sağladığı nesne sınıfı örnekleri için başka federeler tarafından gelebilecek güncelleme istekleri durumunda RTI tarafından federe üzerinden çağrılacak metottur. RTI, bu metodu çağırırken federeye simüle ettiği hangi nesne sınıfı örneği için değer güncellemesi yapması gerektiğine ait parametreleri de sağlamaktadır.
- *reflectAttributeValues*: Federe tarafından abone olunan ve keşfi yapılmış nesne örnekleri için bu örneği simüle eden federe tarafından güncelleme yapılması durumunda, bu güncellemelerin federeye yansıtılmasını sağlayan metottur. Metoda parametre olarak güncellemesi yapılmış özelliklerle bu özelliklerin yeni değerleri kodlanmış olarak aktarılmaktadır.
- *receiveInteraction*: Federenin abone olduğu bir etkileşim sınıfının simülasyonda gönderilmesi durumunda bu etkileşim sınıfının ve parametrelerinin kodlanmış olarak federeye aktarılacağı metotdur. Federe Elçisi, simülasyondaki her etkileşim sınıfını, RTILink katmanında bulunan Simülasyon Mesajı tipinde bir değişkenle temsil etmektedir.
- *removeObjectInstance*: Federe tarafından keşfedilmiş ve güncellemeleri bu federeye aktarılmakta olan bir nesne sınıfı örneğinin simülasyon ortamından silinmesi durumunda RTI'ın federe tarafında çağıracağı metottur.

Simülasyon Mesajı sınıfı, simülasyon üzerinden gönderilen etkileşim sınıflarının uygulama katmanına tek bir ata sınıf üzerinden gönderilebilmesi için kullanılmaktadır. Bu sınıf, etkileşim sınıfı verisinin kodlanmış haline karşı gelen dizi yapısını içermektedir. Üst katmanda çözümleme işlemi bu yapıya uygulanacaktır. Tersi durumda üst katmandan gelen veri kodlama işleminden sonra bu yapı içinde saklanacaktır.

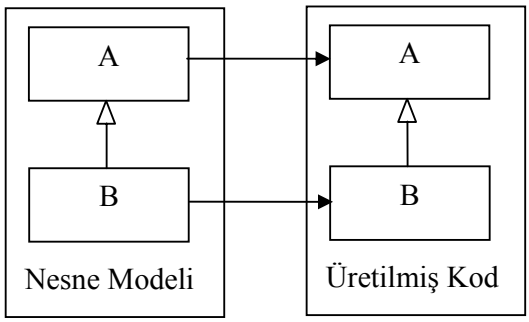
Simülasyon Nesnesi sınıfı, simülasyon üzerinden alınan nesne güncellemelerinin uygulama katmanına tek bir sınıf üzerinden gönderilebilmesi için tasarlanmıştır. Bu sınıf, nesne sınıfı verisinin kodlanmış haline karşı gelen dizi yapısını içermektedir. Üst katmanda çözümleme işlemi bu yapıya uygulanacaktır. Tersi durumda üst katmandan gelen veri kodlama işleminden sonra bu yapı içinde saklanacaktır.

4.2 OMGenerator (Object-Model Generator)

Federenin ya da tüm federasyonun nesne modelinden etkileşim ve nesne sınıflarına karşı gelen C++ sınıflarını, bu sınıfların RTI'a gönderilme ve alınmaları esnasında kullanılacak olan kodlayıcı ve çözücü sınıfları üreten uygulamadır. Bu uygulama Ruby [11] programlama dili ve üretilecek sınıflara ait şablon dosyaları kullanılarak oluşturulmuştur. Kod üretiminde Ruby dilinin tercih edilmesinin sebepleri arasında, bu programlama dilinin açık kaynak kodlu olarak geliştirilmiş olması ve bir çok Linux dağıtımında bulunması, nesne yönelimli bir dil olması ve sunduğu hizmetleri nesneler üzerinde sağladığından türetme yolu ile sağlanan hizmetlerin değiştirilebilmesine imkan vermesi, çalışma kapsamında ihtiyaç duyulan XML ve dosya işlemleri için oldukça yetenekli kütüphaneler içermesi sayılabilir.

Nesne ve etkileşim sınıfları içerisinde yer alan özellikler ve parametreler, nesne modeli içerisinde tanımlanan veri tipleri ile temsil edilirler. Bu nedenle, aynı veri tipine sahip iki verinin kodlama ve çözme işlemleri aynı olmaktadır. Bu benzerlik, üretilen kodlama ve çözme metotlarının verilerin temsil edildiği veri tipleri için oluşturulmasını ve gerekli durumlarda kullanılmasını mümkün kılmıştır. Böylece bir veri tipi için üretilen kodlama ve çözme metotları bu veri tipiyle temsil edilen her veri için tekrar kullanılabilecek şekilde tasarlanmıştır.

Nesne ve etkileşim sınıfları ile kodlayıcı ve çözücü sınıfların üretilmesinde bu sınıflar arasında nesne model şablonunda tanımlanmış olan hiyerarşik yapıdan faydalanılmaktadır. Model şablonunda A sınıfından türemiş bir B sınıfı için üretilen C++ sınıfı da A sınıfı için üretilmiş sınıftan türemektedir. Bu durum Şekil 4.3'de gösterilmiştir.



Şekil 4.3 : Üretilmiş kod hiyerarşisi.

Bundan sonraki bölümlerde ilk olarak hem nesne hem de etkileşim sınıflarının kodlayıcı ve çözücü sınıflarında kullanılmak üzere veri tipleri için kodlama ve çözme metotlarının üretilmesi ele alınmış, daha sonra nesne ve etkileşim sınıfları ve bunların kodlayıcı ve çözücü sınıflarının üretilmesi ayrı başlıklarda anlatılmıştır.

4.2.1 Veri tipleri için kodlayıcı ve çözücü metotların üretilmesi

Kod üretiminin ilk adımında Nesne Model Şablonu dosyası içerisinde belirtilen veri tipleri için hazırlanan şablonlar kullanılarak bu veri tiplerine ait kodlama ve çözme metotlarının üretilmesi işlemi gerçekleştirilmektedir. Bu amaçla hazırlanan şablonlardan ilki bu metotların gösterimlerinin bulunacağı başlık dosyasının üretilmesi diğeri ise bu metotlara ait tanımlamaların yer aldığı kaynak dosyanın üretilmesi amacıyla hazırlanmıştır.

Kod üretimi işleminde diğer üretilen kodlar nesne yönelimli olduğu halde, bu aşamada veri tipleri için üretilecek kodlar bu nesne sınıflarının içerisinden çağrılmak sureti ile kullanılacaklarından dolayı sınıf bağımsız global metotlar olarak tanımlanmıştır.

Üretilecek metotların gösterimlerinin bulunduğu şablon dosyasının basit veri tipleri için içeriği Çizelge 4.1'de verilmiştir.

Çizelge 4.1 : Basit veri tipi için şablon metot gösterimi.

```
<%basicTypes.each {|attribute|%>

void encode<%=attribute.dataType%> (ByteArray&, <%=attribute.dataType%>&);

void decode<%=attribute.dataType%> (<%=attribute.dataType%>&, const void *,
int);
<%}%>
```

Buna göre her basit veri tipi için bir adet kodlama ve bir adet çözme metodu tanımlanmaktadır. Kodlama metodu gösterimi ilk parametre olarak verinin kodlanacağı bayt dizisine referans alırken, ikinci parametre olarak kodlanacak veri tipinde bir referans almaktadır. Çözme metodu gösterimi ise ilk parametre olarak çözülen değerin atanacağı veri tipinden bir referans alırken, ikinci parametre olarak kodlanmış değerin bulunduğu bir dizi işaretçisi ve son parametre olarak bu dizinin boyutunu belirten bir değer almaktadır. Her veri tipi için oluşturulan bu metot gösterimleri bir başlık dosyası içerisinde üretilmektedir.

Bir diğer şablon dosyası gösterimleri yapılan kodlama ve çözme metotlarının tanımlarının üretilmesi amacıyla kullanılmaktadır. Bu tanımların bulunduğu şablon dosyasının basit veri tipleri için içeriği Çizelge 4.2'de verilmiştir.

Çizelge 4.2 : Basit veri tipi için şablon metot tanımı.

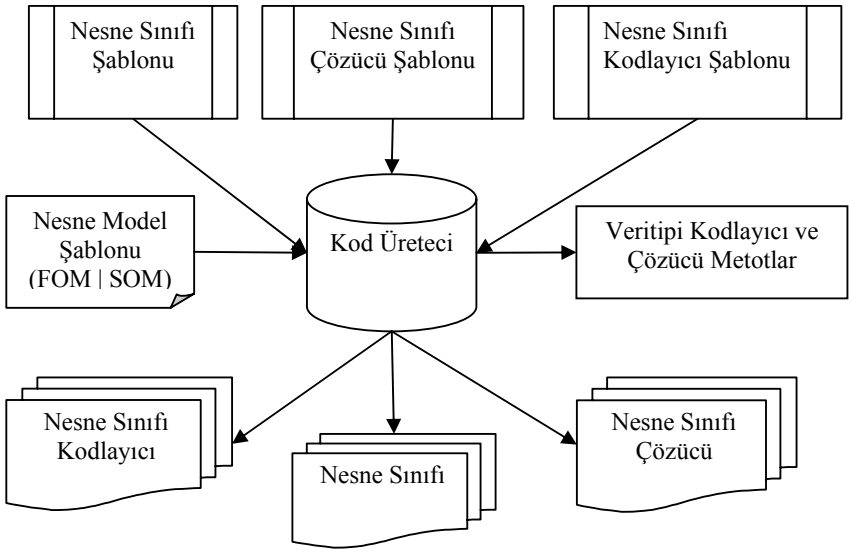
```
<%basicTypes.each {|attribute|%>
void encode<%=attribute.dataType%> (ByteArray& netvec, <%=attribute.dataType%>&
netval)
{
        netvec = netval.toByteArray();
}

void decode<%=attribute.dataType%> (<%=attribute.dataType%>& netval, const void *pAny,
int pSize)
{
        ByteArray netvector;
        netvector.resize(pSize);
        std::memcpy(&netvector[0], pAny, pSize);
        ByteWrapper wrapper(netvector);
        netval.decode(wrapper);
}
<%}%>
```

Kodlama metodu içerisinde veri tipi için gerekli dizilim baytlarının da yerleştirilerek verinin bir bayt dizisi olarak temsil edilmesini sağlayan *toByteArray* metodu çağrılarak bu değer parametre olarak geçilen bayt dizisi tarafından referans gösterilmiştir. Çözme metodunda ise gerekli boyut kadar yeniden boyutlandırılan bayt dizisine alınan kodlanmış değer yine veri tipi için tanımlanmış olan çözücü metoda (*decode*) parametre olarak verilerek çözülmüş değerin parametre olarak geçilen veri tipindeki değişkene atanması sağlanmıştır. Her veri tipi için oluşturulan bu metot tanımları bir kaynak dosya içerisinde üretilmektedir.

4.2.2 Nesne sınıflarının, kodlayıcı ve çözücü sınıfların üretilmesi

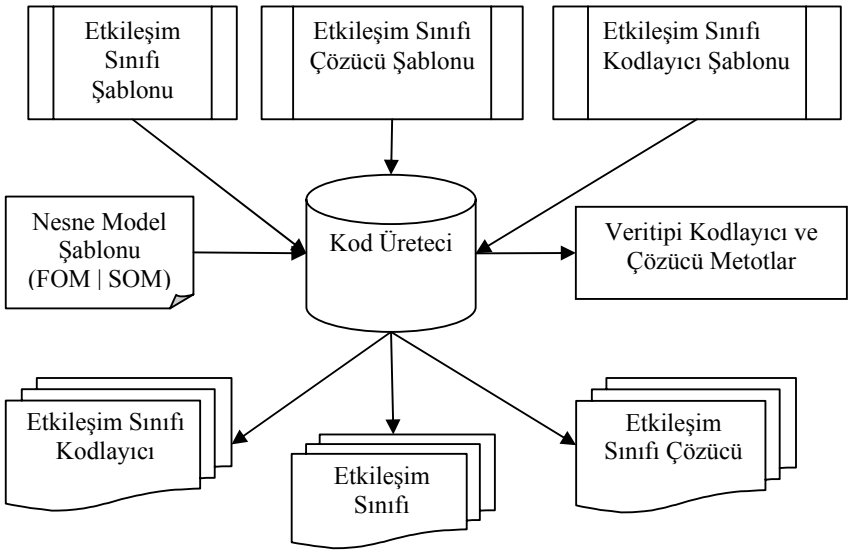
Nesne sınıflarının, kodlayıcı ve çözücü sınıfların üretilmesinde, bu sınıflar için hazırlanan şablon dosyaları kullanılarak bunlar için ayrı ayrı C++ sınıflarının üretilmesi sağlanmıştır. Kodlayıcı sınıflar, bu C++ nesne sınıflarının üye değişkenlerini RTILink katmanında yer alan Simülasyon Nesnesi sınıfına dönüştürürken, çözücü sınıflar, bu işlemin tersini yaparak RTILink katmanındaki Simülasyon Nesnesi sınıfından nesne sınıfının üye değişkenlerini oluşturur. Nesne modelinde yer alan her veri tipi için ise bu tipteki verilerin kodlanması ve çözülmesi işlemlerinde kullanılabilecek kodlayıcı ve çözücü metotlar üretilmektedir. Şablonların bu amaçla kullanımı Şekil 4.4'de gösterilmiştir:



Şekil 4.4 : Nesne sınıfı üretimi.

4.2.3 Etkileşim sınıflarının, kodlayıcı ve çözücü sınıfların üretilmesi

Etkileşim sınıflarının, kodlayıcı ve çözücü sınıfların üretilmesinde, bu sınıflar için hazırlanan şablon dosyaları kullanılarak bunlar için ayrı ayrı C++ sınıflarının üretilmesi sağlanmıştır. Kodlayıcı sınıflar, bu C++ etkileşim sınıflarının üye değişkenlerini RTILink katmanında yer alan Simülasyon Mesajı sınıfına dönüştürürken, çözücü sınıflar, bu işlemin tersini yaparak RTILink katmanındaki Simülasyon Mesajı sınıfından etkileşim sınıfının üye değişkenlerini oluşturur. Nesne modelinde yer alan her veri tipi için ise bu tipteki verilerin kodlanması ve çözülmesi işlemlerinde kullanılabilecek kodlayıcı ve çözücü metotlar üretilmektedir. Şablonların kullanımı Şekil 4.5'de gösterilmiştir:



Şekil 4.5 : Etkileşim sınıfı üretimi.

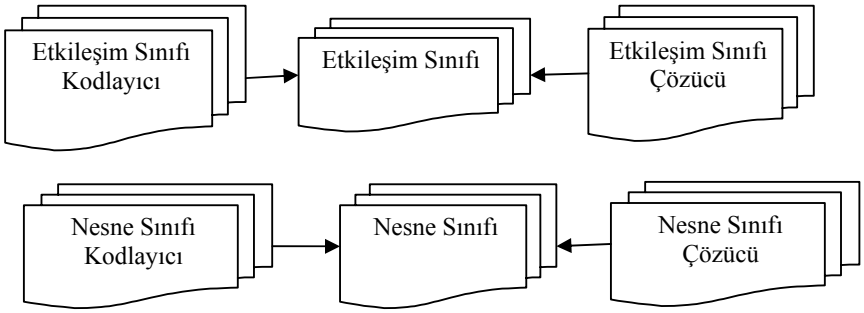
Veri tipleri için üretilen kodlayıcı ve çözücü sınıflar veriden bağımsız olduklarından hem etkileşim sınıflarının parametreleri hem de nesne sınıflarının özellikleri için ortak olarak kullanılmaktadır.

4.3 FOMLink Katmanı

OMGenerator uygulaması tarafından üretilen C++ dosyalarından oluşturulan katmandır. Kod üretici uygulamanın oluşturduğu kaynak dosyalar bu katmanda birleştirilerek kütüphane olarak sunulmaktadır.

Nesne sınıfları, nesne modelinde bulunan nesne sınıflarının C++ dilindeki sınıf karşılıklarıdır. Bu sınıflar, nesne modelinde bulunan sınıf özelliklerinin tiplerine uygun olan üye değişkenlerle, bu üye değişkenler için erişim ve değiştirici metotlar içermektedir.

Etkileşim sınıfları, nesne modelinde bulunan etkileşim sınıflarının C++ dilindeki sınıf karşılıklarıdır. Bu sınıflar, nesne modelinde bulunan sınıf parametre tiplerine uygun olan üye değişkenlerle, bu üye değişkenler için erişim ve değiştirici metotlar içermektedir. Şekil 4.6, bu katmanda bulunan sınıfların ilişkisini göstermektedir.



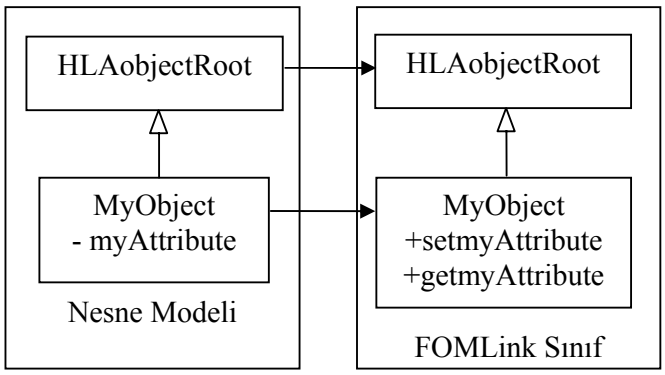
Şekil 4.6 : FOMLink katmanı.

4.4 FOMMapper Katmanı

Geliştirilen platformun FOM değişikliklerinden etkilenmesinin en alt seviyede tutulması amacıyla gerçekleştirilmiş kütüphanedir. Bu kütüphane, temel olarak nesne ve etkileşim sınıfları ile bu sınıflara karşı gelen çözücü ve kodlayıcı sınıfların eşleştirilmelerinin yapıldığı bölümdür. Belirli bir nesne modeli için üretilen tüm nesne ve etkileşim sınıfları ile bunların çözücü ve kodlayıcı sınıfları bu kütüphanede yer alan fabrika sınıflarına kaydettirilmektedir. Gerekli durumlarda, bir sınıfa karşı gelen kodlama ya da çözücü sınıfın bulunması bu eşlemeler üzerinden gerçekleştirilmektedir. FOMLink katmanında üretilen tüm sınıflar için bu kayıt işlemi, uygulama katmanında ilkleme işlemleri esnasında gerçekleştirilmektedir.

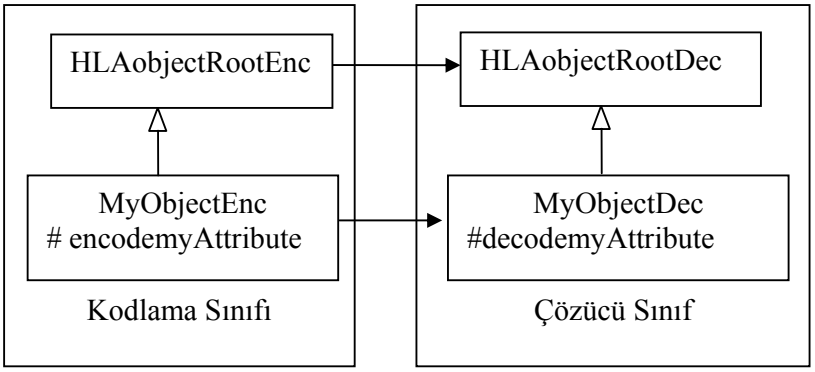
FOM-çeviklik açısından bu katmanın kullanımı, FOMLink üretimi sırasında kullanılan veri tipi temelli çözücü ve kodlayıcı sınıfların üretilmesi özelliği ile sağlanmıştır. Buna göre kullanıcı, nesne modeline yeni nesne ve etkileşim sınıflarının eklenmesi durumunda, tüm nesne modeli için tekrar kod üretilmesi yerine sadece bu sınıflar için gerekli eşleme işlemlerini FOMMapper kütüphanesi içerisinde sunulan fabrika sınıflarına kayıt ettirerek kütüphaneyi geliştirebilecektir. Fabrika sınıfları kullanıcıya bu amaca yönelik kayıt servisleri sunmaktadır.

Kullanıcı bu amaçla eklemek istediği sınıfa ait yeni sınıfı hiyerarşide uygun olan sınıftan türeterek gerekli özellik ya da parametreleri de bu sınıfa ekler. Bu üye değişkenler için aynı zamanda sınıfa erişim ve kurucu metotlar da eklenir. Şekil 4.7'de *HLAobjectRoot* sınıfından türeyen ve *myAttribute* isimli özellik içeren *MyObject* nesne sınıfının katmana eklenmesi için türetilmesi gereken sınıflar gösterilmiştir.



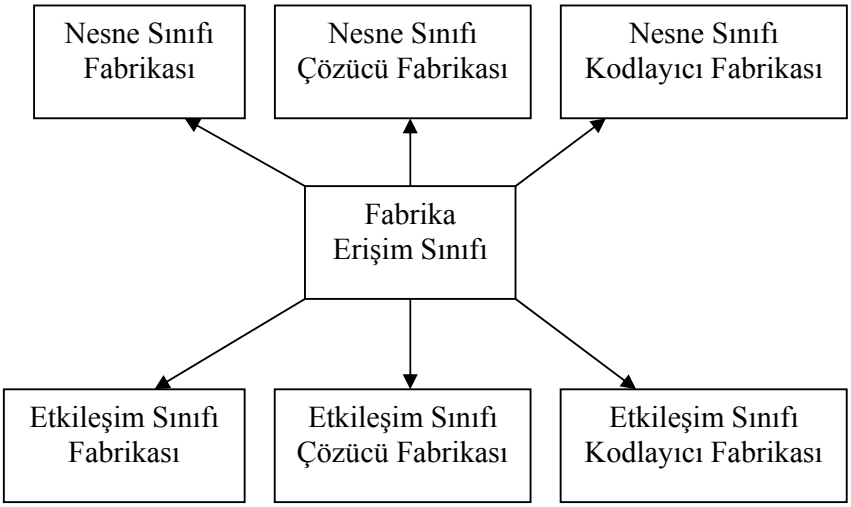
Şekil 4.7 : Nesne sınıfı türetme.

Çözücü ve kodlayıcı sınıflar da uygun sınıflardan türetilerek eklenen özellik ya da parametreler için çözücü ve kodlayıcı metotlar bu sınıflara eklenir. Şekil 4.8'de bu durum gösterilmiştir.



Şekil 4.8 : Kodlama ve çözücü sınıf türetme.

Yeni eklenen sınıflar için gerçekleştirilecek kodlama ve çözme metotları, bu nesnelerin özellik ya da parametrelerinin veri tiplerine bağlı olarak değişecektir. Bu veri tiplerinin, nesne modelinde mevcut veri tiplerinden olması durumunda kullanıcı geliştireceği kodlayıcı ve çözücü sınıflarda bu veri tipi için üretilmiş metotları kullanabilecektir. Bu şekilde kullanıcının eklemesi gereken kodun azaltılması sağlanmıştır. Şekil 4.9'da bu katmanda yer alan fabrika sınıfları ile bunlara tek noktadan erişimi mümkün kılan erişim sınıfı arasındaki ilişki gösterilmiştir.



Şekil 4.9 : FOMMapper katmanı.

4.5 SIMLink Katmanı

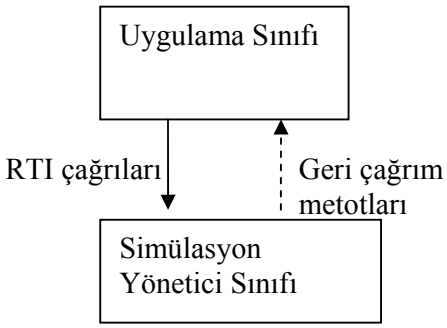
Simülasyon uygulamasına servis sağlayan en üst düzey katmandır. Bu katmanda uygulamanın RTI ile haberleşmesini sağlayan yönetici sınıflar bulunmaktadır. Uygulama, simülasyon ortamından ihtiyaç duyduğu verileri bu yönetici sınıflar vasıtası ile alabilmekte ve yine üretilen bilgileri simülasyon ortamına yayınlayabilmektedir. Bu katmanın sunduğu hizmetler aşağıdaki başlıklarda özetlenebilir:

- Etkileşim sınıflarına geri çağrım metotları kaydettirme.
- Nesne kayıtlarına, güncellemelerine ve silinmelerine geri çağrım metodu kaydettirme.
- Etkileşim sınıfı gönderebilme.
- Nesne kaydetme, güncelleme ve silme.

Simülasyon ortamından uygulama katmanına bilgi akışı, SIMLink katmanına kayıt ettirilen geri çağrım metotları ile mümkün olmaktadır. Buna göre uygulama, SIMLink katmanında bulunan yönetici sınıf üzerinden, simülasyon ortamından almak istediği bilgiler için metotlar kaydettirebilmekte, simülasyon ortamından kayıt olunan tipte etkileşim veya nesne güncellemesi geldiğinde ise yönetici sınıf, kayıt ettirilen bu uygulama metodunu gelen bilgiyi parametre olarak geçmek suretiyle çağırmaktadır.

Uygulama düzeyinden simülasyon ortamına bilgi akışında ise, yayınlanmak istenen etkileşim veya nesne sınıflarına karşı gelen C++ sınıflarının üye değişkenlerine ilgili değer atamaları yapılarak, SIMLink katmanında bulunan yönetici sınıf üzerinden bu veriler RTI'a yayınlanmaktadır.

Şekil 4.10, uygulama sınıfı ile SIMLink katmanında bulunan yönetici sınıf arasındaki bağlantıyı göstermektedir.



Şekil 4.10 : SIMLink katmanı.

Bu katmanda gerçeklenen sınıfları ve katmanın sunduğu hizmetler bakımından sınıfların değerlendirilmesi aşağıda sunulmuştur.

Simülasyon Yönetici sınıfı katmanda yer alan temel sınıftır. Federenin simülasyon arayüzünün katmanda bulunan diğer sınıflar kullanılarak sağlanmasından sorumludur. Uygulamada çeşitli noktalardan ihtiyaç duyulabileceği ve uygulamanın simülasyon arayüzünün tek olması gerektiğinden tekil bir sınıf olarak tasarlanmıştır. Sağladığı metotlar aşağıda verilmiştir.

- *getInstance*: Yönetici sınıfın tekil erişim metodudur. Bu metot, ilk çağrıldığında bu tipten bir nesne oluşturmakta ve bu metoda yapılacak diğer bütün çağrılarda bu değişkeni döndürmektedir.
- *initFederate*: Federenin otomatik olarak üretilen nesne ve etkileşim sınıflarının kayıt ettirildiği FOMMapper sınıfını parametre olarak alan ve federenin federasyona giriş işlemlerini daha alt seviye çağrılar yaparak gerçekleştiren metottur.
- *getFederateHandle:* Federeye RTI tarafından atanan tekil tanımlayıcı bilgisini döndüren metottur.
- *addDiscoverObjectCallBack:* Federenin abone olduğu tipte bir nesne sınıfı örneğinin simülasyona kayıt ettirilmesi durumunda çağrılması istenen geri çağrım metodunun kaydının yapıldığı metottur.
- *addUpdateObjectCallBack:* Abone olunan nesne sınıfı örneğinin güncellenmesi durumunda çağrılması istenen geri çağrım metodunun kaydının yapıldığı metottur.

- *addRemoveObjectCallBack:* Federenin abone olduğu tipte bir nesne sınıfı örneğinin simülasyondan silinmesi durumunda çağrılması istenen geri çağrım metodunun kaydının yapıldığı metottur.
- *addInteractionCallBack:* Federe tarafından abone olunan etkileşimin simülasyon ortamından alınması durumunda uygulama katmanında çağrılması istenen metodun kayıt ettirildiği metottur.
- *removeInteractionCallBack:* Etkileşim için kayıt ettirilmiş geri çağrım metodunun kaldırılmasını sağlayan yönetici sınıf metodudur.
- *start*: Yönetici sınıf tarafından sağlanan bu metot, federeye simülasyon ortamından gelecek verilerin aktarımının sağlanması için düzenli olarak çağrılması gereken RTILink katmanı servislerinden *evokeFederateAmbassadorCallbacks* çağrısının periyodik yapılmasını sağlayacak mekanizmanın başlatıldığı metottur. Bu çağrıdan sonra simülasyon ortamından gelen her veri yönetici sınıfa kayıt ettirilmiş uygulama metotlarına yansıtılmaktadır.
- *createAndInitObject:* İsmi parametre olarak verilen nesne sınıfına karşı gelen FOMLink sınıfının oluşturularak ilklendiği ve geri döndürüldüğü metottur. Tüm nesne sınıflarına karşı gelen FOMLink sınıfları bu katmanda yer alan *BaseEntityObject* sınıfından türetildiği için bu metot *BaseEntityObject* tipinden bir işaretçi dönmektedir. Kullanıcı metot tarafından döndürülen işaretçiyi uygun FOMLink sınıfına dönüştürerek kullanabilmektedir.
- *tick:* Parametre olarak geçilen FOMLink sınıfına karşı gelen nesne sınıfı örneğinin güncelleme bilgisini simülasyon ortamına gönderen metottur.
- *deleteObject:* Oluşturulup yayınlanmakta olan simülasyon nesne sınıfı örneğini silmek amacıyla gerçekleştirilen metottur.
- *simSendMessage:* Parametre olarak geçilen FOMLink etkileşim sınıfını simülasyon ortamına, karşı gelen etkileşim sınıfı olarak gönderen metottur.
- *exitFederation:* Federenin federasyondan çıkma işlemlerini gerçekleştiren metot olarak tasarlanmıştır.

Temel Nesne Sınıfı (*BaseEntityObject*), bu katmanda nesne modeli şablonlarındaki nesne sınıflarına karşı gelen sınıf olarak kullanılmaktadır. Simülasyon nesneleri için bu katmanda sağlanan her hizmet bu temel nesne sınıfı üzerinde çalışmaktadır. FOMLink katmanında otomatik olarak üretilen her C++ sınıfı bu temel nesneden türeyecek şekilde tasarlandığından dolayı bu katmanda sağlanan hizmetler, FOMLink katmanında yer alan tüm nesne sınıfları üzerinde geçerli olmaktadır.

Bu temel sınıf aynı zamanda RTILink katmanında yer alan ve nesne sınıfına ait kodlanmış veriyi temsil eden Simülasyon Nesnesi sınıfından bir üye değişkene de sahiptir. Bu üye değişken, abone olma durumunda RTI üzerinden gelen kodlanmış nesne güncellemelerini tutarken, yayınlama durumunda RTI'a gönderilecek olan verinin yine kodlanmış halini saklamak için kullanılmaktadır. FOMLink katmanında otomatik olarak üretilen ve bu sınıftan türeyen C++ sınıfları ise yine FOMLink katmanında bulunan kodlayıcı ve çözücü sınıfları da kullanarak uygulama katmanı ile RTI arasında bu verinin dönüşümünü gerçekleştirmektedir. FOMLink katmanında üretilen C++ sınıfları bu sınıfa, nesne modelindeki her özelliğe karşı gelen bir üye değişkenle bu üye değişken için erişim ve değişim metotlarını eklemektedir.

Sınıf, ayrıca Simülasyon Yönetici sınıfına bir işaretçi içerdiğinden her sınıf kendini direkt olarak simülasyon ortamına yayınlayabilme özelliğine de sahiptir. Sınıfın sağladığı metotlar aşağıda verilmiştir.

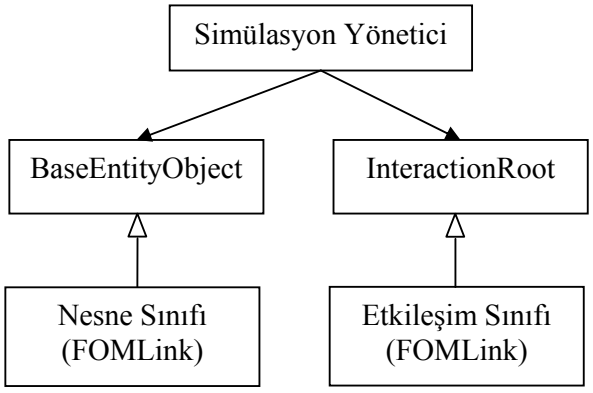
- *setEntityDataRep*: Sınıf içerisinde bulunan ve RTILink katmanında yer alan Simülasyon Nesnesi tipinden üye değişken için kurucu metottur.
- *getEntityDataRep*: Sınıf içerisinde bulunan ve RTILink katmanında yer alan Simülasyon Nesnesi tipinden üye değişken için erişim metodudur.
- *setAttribute:* Sınıfın bir üye değişkenini boyutuna uygun olarak RTILink katmanındaki Simülasyon Nesnesi sınıfına kodlanmış halde ekleyen metottur.
- *getAttribute:* Sınıfın bir üye değişkenini boyutuna uygun olarak RTILink katmanındaki Simülasyon Nesnesi sınıfından kodlanmış halde döndüren metottur.
- *tick:* Simülasyon ortamına nesne güncellemesinin gönderilmesini sağlayan metottur. Sahip olunan Yönetici Sınıf üzerinden bu güncelleme gönderilmektedir.

Temel Etkileşim Sınıfı (*InteractionRoot*), bu katmanda nesne modeli şablonlarındaki etkileşim sınıflarına karşı gelen sınıf olarak kullanılmaktadır. Etkileşimler için bu katmanda sağlanan her hizmet bu temel etkileşim sınıfı üzerinde çalışmaktadır. FOMLink katmanında otomatik olarak üretilen her C++ sınıfı bu temel etkileşimden türeyecek şekilde tasarlandığından dolayı bu katmanda sağlanan hizmetler, FOMLink katmanında yer alan tüm etkileşim sınıfları üzerinde geçerli olmaktadır.

Sınıf, ayrıca Simülasyon Yönetici sınıfına bir işaretçi içerdiğinden her sınıf kendini direkt olarak simülasyon ortamına gönderebilme özelliğine de sahiptir. Sınıfın sağladığı metotlar aşağıda verilmiştir.

- *setParameter:* Sınıfın bir parametresini boyutuna uygun olarak RTILink katmanındaki Simülasyon Mesajı sınıfına kodlanmış halde ekleyen metottur.
- *getParameter:* Sınıfın bir parametresini boyutuna uygun olarak RTILink katmanındaki Simülasyon Mesajı sınıfından kodlanmış halde döndüren metottur.
- *tick:* Simülasyon ortamına etkileşimin gönderilmesini sağlayan metottur. Sahip olunan Yönetici Sınıf üzerinden bu işlem gerçekleştirilmektedir.

SIMLink katmanında, Simülasyon Yönetici sınıfı sağladığı hizmetleri Temel Nesne Sınıfı ve Temel Etkileşim Sınıfı içerisinde bulunan *virtual* metotları kullanarak gerçekleştirdiğinden ve FOMLink katmanında otomatik olarak üretilen nesne ve etkileşim sınıfları bu temel sınıflardan türediklerinden dolayı, yönetici sınıf FOMLink katmanında yer alan tüm sınıfları da yönetebilmektedir. Şekil 4.11, nesne ve etkileşim sınıflarının Yönetici Sınıf ile ilişkisini göstermektedir.



Şekil 4.11 : Nesne ve etkileşim yönetimi.

4.6 Mimari Karşılaştırma

Geliştirilen mimarideki katman yaklaşımı hem GENESIS hem de VR-Link platformuyla benzerlik taşımaktadır. Buna göre her üç platform da federelerin RTI ile bağlantılarının sağlanacağı bir bağlantı katmanı ile kullanılan nesne ve etkileşim sınıflarının bulunduğu kütüphaneler sunmaktadır. En üst düzeyde ise uygulama kodu tarafından kullanılacak servisleri sağlayan yönetici birer katman içermektedirler.

Çizelge 4.3'de geliştirilen katmanların diğer platformlarda karşılığı olan katmanlar ya da sınıflar verilmiştir.

Çizelge 4.3 : Katman eşleştirme.

Katman İsmi	**GENESIS**	**VR-Link**
RTILink	Virtual RTI Lib	DIS/HLA Interface
FOMLink	Datatypes, objects,interactions, synchro. lib	Nesne ve etkileşim kütüphaneleri
OMGenerator	GENESIS Tool	VR-Link Code Generator
FOMMapper	-	FOMMapper kütüphanesi
SIMLink	GENESIS Lib	Publisher ve Reflected Object List sınıfları

4.7 Kod Üretimi Karşılaştırma

Kod üretim yetenekleri açısından bakıldığında, GENESIS platformu girdi olarak GenDL biçimine uygun girdi dosyaları alarak federe için nesne model dosyasını ve sınıf kütüphanelerini üretirken, VR-Link bu sınıfları, mevcut nesne modeli dosyasından üretebilmekte ve ek bir girdiye ihtiyaç duymamaktadır. Geliştirilen OMGenerator uygulaması da VR-Link ile benzer bir yaklaşım kullanarak etkileşim ve nesne sınıflarını ve bunlara ait çözücü ve kodlayıcı sınıfları nesne modelinden üretmektedir.

Diğer taraftan kodlayıcı ve çözücü sınıfların üretilmesi amacıyla, VR-Link ticari uygulamasının sağlamış olduğu “Code Generator” uygulaması, nesne modelindeki karmaşık veri tipleri için kodlayıcı ve çözücü sınıflar üretememektedir. OMGenerator uygulaması nesne modelinde tanımlı tüm veri tipleri için bu sınıfları üretebilme yeteneğine sahiptir.

FYA ile kod üretme teknikleri açısından kıyaslandığında ise, özellikle kodlama/çözücü metotların oluşturulmasında kullanılan yaklaşım bakımından önemli farklar göze çarpmaktadır. Buna göre FYA, nesne modelinde bulunan her özellik ve parametre için veri tipinden bağımsız kodlama/çözme metotları üretirken, OMGenerator tarafından üretilen kodlama/çözme metotlarının veri tipi temelli olması bu katmanda üretilen kodun daha etkin olmasını ve gereksiz kod tekrarının önlenmesini sağlamıştır.

Üretilen sınıfların uygulama sınıfları ile karışmasının önlenmesi açısından 4 platform da üretilen kodların belirli bir isim uzayında oluşturulmasına imkan sağlamaktadır.

Çizelge 4.4'de bu uygulamaların kod üretim yaklaşımları kıyaslanmıştır.

Çizelge 4.4 : Kod üretim yaklaşımları.

Katman	**Hedef Dil**	**Model**	**Girdi**	**Çıktı**
GENESIS Tool	C++	Katman Üretimi	GenDL nesne ve etkileşim tanım dosyaları	Nesne Modeli ve nesne, etkileşim, çözücü ve kodlayıcı sınıf kaynak dosyaları
VR-Link Code Generator	C++	Katman Üretimi	Nesne Model Dosyası	Nesne, etkileşim, çözücü ve kodlayıcı sınıf kaynak dosyaları
OMGenerator	C++	Katman Üretimi	Nesne Model Dosyası	Nesne, etkileşim, çözücü ve kodlayıcı sınıf kaynak dosyaları
FYA	C++	Katman Üretimi	Nesne Model Dosyası	Nesne, etkileşim, çözücü ve kodlayıcı sınıf kaynak dosyaları

4.8 Kullanım Karşılaştırma

Mimarilerin nesne güncellemeleri ve bu güncellemelere abone olma şekli ile etkileşim sınıfı gönderme ve alma şekilleri farklılık göstermektedir. Bu kullanım farklılıkları gönderim ve alma işlemleri için ayrı ayrı başlıklarda incelenmiştir.

4.8.1 Nesne güncelleme ve etkileşim gönderme

VR-Link Mimarisinde nesne güncellemeleri her nesne için ayrı yazılan yayınlayıcı (publisher) sınıflar vasıtası ile yapılırken, etkileşim sınıfları ayrı C++ sınıfları olarak tanımlanmışlardır ve uygulamanın simülasyon bağlantısı üzerinden gönderilirler. Bu akışı sağlayan basit bir kod Çizelge 4.5'de verilmiştir.

Çizelge 4.5 : VR-Link nesne ve etkileşim gönderme.

```
DtExerciseConn exConn();

DtEntityPublisher entityPub();
DtEntityStateRepository *esr = entityPub.entityStateRep();
esr->setMarkingText("VR-Link");
entityPub.tick();

DtFireInteraction inter;
exConn.sendStamped(inter);
```

Kodda bulunan DtExerciseConn tipindeki nesne, federenin simülasyon ortamına bağlantısını sağlayan sınıf olarak kullanılmaktadır. VR-Link katmanında uygulamanın mimari bağımsız bağlantı arayüzü olarak kullanılmaktadır. Tüm nesne güncellemeleri ve etkileşim gönderme işlemleri bu sınıf tarafından gerçekleştirilmektedir. DtEntityPublisher tipindeki sınıf, benzetimi yapılacak olan nesne modeli sınıfının yayınlayıcı sınıfı olarak kullanılmış olup, içerisinde nesne sınıfının özelliklerinin tutulduğu bir durum sınıfı (DtEntityStateRepository) içermektedir. Kullanıcı, yayınlayıcı sınıf üzerinden erişebildiği bu durum sınıfının kurucu metotlarını (setMarkingText vb.) kullanarak nesnenin simülasyon ortamına yayınlanacak özelliklerini doldurabilmektedir. Özellikleri bu yolla kurulmuş olan sınıf örneği yayınlayıcı sınıf tarafından sağlanan tick metodu vasıtası ile simülasyon ortamına gönderilmektedir. Bu gönderim amacıyla metot içerisinde federenin simülasyon bağlantısı olan DtExerciseConn sınıfı kullanılmaktadır.

Etkileşim gönderme işlemi ise, etkileşime karşı gelen C++ sınıfı (*DtFireInteraction*) tipinde bir nesne oluşturularak ve bu nesnenin sağladığı kurucu metotlar vasıtası ile etkileşim parametreleri kurularak sağlanmaktadır. Bu sınıf daha sonra simülasyon bağlantı sınıfı olan *DtExerciseConn* tarafından sağlanan ve etkileşim gönderme işlemini gerçekleştiren *sendStamped* metoduna parametre olarak verilerek simülasyon ortamına yayınlanmıştır.

Geliştirilen mimaride ise nesne güncellemeleri bu nesnelere karşı gelen C++ nesneleri üzerinden herhangi bir yayınlayıcı sınıf olmaksızın gerçekleştirilir. Her nesneye parametre olarak geçilen simülasyon yönetici sınıfı vasıtasıyla bu işlem mümkün olmaktadır. Benzer şekilde etkileşim sınıfları da parametre olarak geçilen simülasyon yöneticisi sayesinde kendilerini simülasyon ortamına gönderme özelliğine sahiptirler. Bu akışı sağlayan basit bir kod Çizelge 4.6'da verilmiştir.

Çizelge 4.6 : SIMLink nesne ve etkileşim gönderme.

```
SimManager  manager;
MyObject  *ptr2;

ptr2=(MyObject*)manager.createAndInitObject(name,"MyObject1");

manager.initFederate(initFom());
manager.start();
ptr2->tick();

MyInteraction inter(&manager);
inter.send();
```

FOMLink katmanı VR-Link ürününden farklı olarak nesne sınıfları için yayınlayıcı sınıflar kullanmamakta ve nesnelerin durum bilgilerini ayrı durum sınıflarında değil nesne sınıflarının içerisinde üye değişkenler olarak saklamaktadır. SIMLink katmanında yer alan SimManager sınıfı VR-Link katmanında bulunan DtExerciseConn sınıfına benzer olarak federenin simülasyon bağlantısını sağlamaktadır. Bu sınıf, aynı zamanda nesne modelinde yer alan sınıf isimlerini parametre olarak alan ve bu sınıfa karşı gelen FOMLink katmanındaki C++ sınıfını üreten metoda(createAndInitObject) da sahiptir. Örnek kodda yönetici sınıfın bu metodu kullanılarak nesne modelindeki MyObject nesne sınıfı için C++ sınıfı oluşturulmuş ve bir işaretçiye atanmıştır. Oluşturulan bu sınıf, FOMLink katmanında otomatik olarak üretilen sınıf olup nesne sınıfının özellikleri için kurucu metotlar içermektedir. Yönetici sınıf üzerinden çağrılan initFederate metodu kullanılacak olan nesne ve etkileşim sınıflarıyla bunların çözücü ve kodlayıcı sınıflarının fabrika sınıflarına kayıt ettirildiği metottur.

Bu çağrıdan sonra yönetici sınıf üzerinde *start* metodu çağrılarak yönetici sınıfın simülasyon ortamından gelen bilgileri alabilmesi için gerekli geri çağrım (*Callback*) mekanizması başlatılmaktadır. Bu mekanizma, federe tarafından kayıt olunan nesne ve etkileşim sınıfları simülasyon ortamından RTILink katmanı tarafından alındığında bunlara karşı gelen kodlanmış veriyi içeren simülasyon nesnesi ve simülasyon mesajı sınıflarının çözülerek FOMLink sınıflarının üretilmesi ve doldurulması işlemlerini gerçekleştirir. Aynı zamanda uygulama katmanında bu bilgiler için federe tarafından yönetici sınıfa kayıt ettirilmiş geri çağrım metotları varsa oluşturulan bu FOMLink sınıflarını bu geri çağrım metotlarına ileterek simülasyon verisinin uygulama katmanına aktarılması işlemini gerçekleştirir.

Yönetici sınıf üzerinden *start* metodu ile başlatılan geri çağrım mekanizmasından sonra, oluşturulan nesne sınıfı simülasyon ortamına *tick* metodu kullanılarak gönderilebilmektedir. Etkileşim sınıflarına karşı gelen FOMLink sınıfları ise, yönetici sınıf tarafından değil kullanıcı tarafından normal bir C++ değişkeninin tanımlandığı şekilde oluşturulabilmektedir. Bu sınıflar, oluşturulmaları esnasında yönetici sınıfı parametre olarak almaktadırlar. Sınıfların simülasyon ortamına gönderilmesi işlemi, etkileşim sınıfının *send* metodu kullanılarak gerçekleştirilmektedir.

4.8.2 Nesne güncellemesi ve etkileşim alma

VR-Link Mimarisinde nesne güncellemelerinin uygulama katmanına aktarılması yansıtılmış nesne sınıfları (reflected object) ve bu sınıflarını içeren listeler (reflected object list) üzerinden sağlanmaktadır. Etkileşim alma ise her bir etkileşim sınıfına, uygun imzaya sahip bir geri çağrım metodunun kaydettirilmesi ile sağlanmıştır. Bu akışı sağlayan basit bir kod Çizelge 4.7'de verilmiştir.

Çizelge 4.7 : VR-Link nesne ve etkileşim alma.

```
DtExerciseConn exConn();

DtFireInteraction::addCallback(&exConn, staticfireCb, NULL);
DtReflectedEntityList rel(&exConn);

exConn.drainInput();

DtReflectedEntity *first = rel.first();
DtEntityStateRepository *esr = first->entityStateRep();
```

Kodda ilk olarak etkileşim sınıflarının uygulama katmanı tarafından alınabilmesi amacıyle geri çağrım metodu kaydettirilmesi gösterilmiştir. Bu kayıt işlemi VR-Link tarafından her etkileşim için statik olarak sağlanmış ve etkileşimin örneğinden (instance) bağımsız olan addCallback metodu ile sağlanmıştır. Metot parametre olarak federenin simülasyona bağlantı arayüzü olan DtExerciseConn sınıfı tipinde bir işaretçi, bu etkileşimin alınması durumunda uygulama katmanından çağrılması istenen fonksiyona işaretçi (staticfireCb) ve kullanıcı tarafından belirlenen herhangi bir tipe ait başka bir işaretçi almaktadır.

VR-Link katmanında nesne güncellemeleri ise bu katman tarafından yönetilen yansıtılmış nesne listeleri (*DtReflectedEntityList*) üzerinde tutulmaktadır. Bu listeler ilkleme parametresi olarak simülasyon bağlantısının işaretçisini almaktadır.

Simülasyon bağlantısı (*DtExerciseConn*) üzerinden çağrılan *drainInput* metodu simülasyon ortamındaki bilgilerin federeye aktarılması için gerekli işlemlerin yapıldığı en önemli metottur. Bu metot RTI'a, simülasyonda federenin abone olduğu verilerin federeye iletilmesini sağlayacak olan metotların çağrılması için gerekli zamanın tanınması işlemidir. RTI, tüm nesne güncellemelerini ve etkileşimleri federeye bu çağrı içinde aktarmaktadır. VR-Link katmanında bu metodu çağırmayan federelerin simülasyon ortamından bilgi almaları mümkün değildir. Etkileşim sınıflarının ve nesne güncellemelerinin bu metot çağrısı ile birlikte alınması, etkileşim sınıflarına kayıt ettirilmiş olan uygulama katmanı metotlarının çağrılması işlemi ile yansıtılmış nesne listelerinin en güncel verilerle doldurulması işlemlerinin de yine bu metot içinde gerçekleştirilmesini sağlamaktadır.

Örnekte, *drainInput* metodu çağrısından sonra kullanıcı nesne listesi üzerindeki herhangi bir yansıtılmış nesne üzerinden (*DtRefectedEntity*) durum sınıfına (*DtEntityStateRepository*) erişerek ve bu sınıftaki erişim metotlarını kullanarak nesne özelliklerine erişebilmektedir. Simülasyon ortamında bulunan her nesne sınıfı, yansıtılmış nesne listesinde bir elemana karşı gelmektedir.

Geliştirilen mimaride ise, nesne güncellemeleri ve etkileşimler için kayıt ettirilecek geri çağrım metotlarının statik olma zorunluluğu yoktur. Gerekli imzaya sahip herhangi bir sınıf üye metodu yönetici sınıfa nesne güncellemesi ya da etkileşim alma geri çağrımı olarak kaydettirilebilmektedir. Bu metodun çağrımı ilgili simülasyon nesnesinin güncellemesi ya da etkileşim alınması durumunda simülasyon yöneticisi tarafından otomatik olarak gerçekleştirilmektedir. Bu akışı sağlayan basit bir kod Çizelge 4.8'de verilmiştir.

Çizelge 4.8 : SIMLink nesne ve etkileşim alma.

```
TestClass testClass;

manager.addDiscoverObjectCallBack<TestClass>("MyObject",testClass,
&TestClass:: updateObject);

manager.addInteractionCallBack<TestClass>("StartResume",testClass,&T
estClass::startFederate);
```

Kodda etkileşim ve nesne güncellemelerinin alınması amacıyla SIMLink katmanında yer alan *SimManager* sınıfına geri çağrım metodu kaydettirilmesi gösterilmiştir. İlk geri çağrım metodu olarak simülasyon ortamında abone olunan belirli bir tipteki (*MyObject*) nesne sınıfının federe tarafından ilk kez alınması (*Discover*) durumunda çağrılması istenen bir uygulama katmanı metodu kayıt ettirilmiştir. Yönetici sınıfın *addDiscoverObjectCallBack* metodu ilk parametre olarak hangi nesne sınıfı için bu çağrım metodunun kaydettirileceğini, ikinci parametre olarak geri çağrım metodunu içeren uygulama nesnesini ve son parametre olarak da bu nesnenin hangi metodunun geri çağrım metodu olarak kaydettirileceğini almaktadır. Bu kayıt işleminin VR-Link katmanından en büyük farkı kayıt ettirilen metodun uygulama katmanında bulunan herhangi bir sınıf örneğinin (instance) metodu olabilmesidir. VR-Link mimarisinde kayıt ettirilecek bu metot, sınıf örneğinden bağımsız statik bir metot olmak zorundadır.

Etkileşim geri çağrım metotları da yönetici sınıf üzerinden *addInteractionCallback* metodu çağrılarak ve etkileşim sınıfının ismi (*StartResume*), metodun bulunduğu sınıf ve fonksiyon işaretçisi parametre olarak geçilerek kayıt ettirilebilmektedirler.

VR-Link katmanında değinilen ve simülasyon verilerinin federeye aktarılması için düzenli olarak çağrılması gereken *drainInput* metodu içinde yapılan işlemler, SIMLink katmanında yönetici sınıf üzerinden çağrılan *start* metodundan sonra başlatılan geri çağrım mekanizmasında periyodik olarak yürütülmekte olup bu amaçla uygulama katmanında ek bir metot çağrılması zorunluluğu bulunmamaktadır.

4.9 Hizmet Karşılaştırma

Uygulama katmanına sunulan hizmetler bakımından, gerçeklenmiş mimariler birbirinden oldukça farklılık göstermektedirler. Koşum Zamanı Altyapı Uzantıları, sunduğu geliştirme dili ve varlık ilişki diyagramlarından federe kodu üretmesi nedeniyle bu konuda en esnek yapıya sahip platformdur. Mimarinin diğer bir önemli özelliği ise bu yolla sağlanan durum akışına kullanıcı tanımlı metotların eklenebilmesine olanak vermesidir.

VR-Link mimarisi, temel olarak DIS ve HLA standartlarına uygun ve bu standartlarda tanımlanmış olan servislerin sağlanmasına yönelik geliştirildiği için zaman yönetimi ve veri dağıtım servisi gibi ileri HLA servislerini kullanma imkanı vermektedir. Sunulan tüm bu mimari bağımsız hizmetlerin yanında en büyük avantajı geliştirilen uygulamanın çoğu durumda FOM bağımsız olarak gerçekleştirilebilmesidir. Bu özellik geliştirilen uygulama kodlarının FOM değişikliklerinden en az seviyede etkilenmesini sağlamaktadır. Diğer taraftan nesne güncellemeleri ve etkileşim alma durumunda kayıt ettirilecek olan geri çağrım metotlarının statik olması zorunluluğu bu mimarinin olumsuz yönleri arasında sayılabilir. Ticari bir ürün olması nedeniyle lisanslama ücretleri göz önüne alındığında bu ürünle geliştirilmiş olan federelerin maliyetleri onbinlerce dolar olabilmektedir.

Geliştirilen platformun en önemli özelliği nesne güncellemeleri ve etkileşim alma durumunda çağrılacak olan metotların statik olma zorunluluğunun bulunmamasıdır. Bu yolla, herhangi bir sınıfa ait metot, belirli bir nesne güncellemesine ya da etkileşime kaydettirilebilmektedir. Böylece simülasyon verilerinin akışının uygulama katmanında doğrudan ilgili sınıfa aktarılması daha kolay hale getirilmiştir. Diğer taraftan temel HLA servislerinin dışında GENESIS ve VR-Link platformlarında desteklenen ileri HLA servisleri uygulama ihtiyaçları doğrultusunda bu servislerin doğrudan HLA çağrıları ile yapılmasının performans açısından daha uygun olacağı değerlendirildiğinden bu çalışmada desteklenmemiştir. Çizelge 4.9'da HLA servisleri açısından platformların uygulamaya sağladığı hizmetler listelenmiştir.

Çizelge 4.9 : Sunulan HLA servisleri.

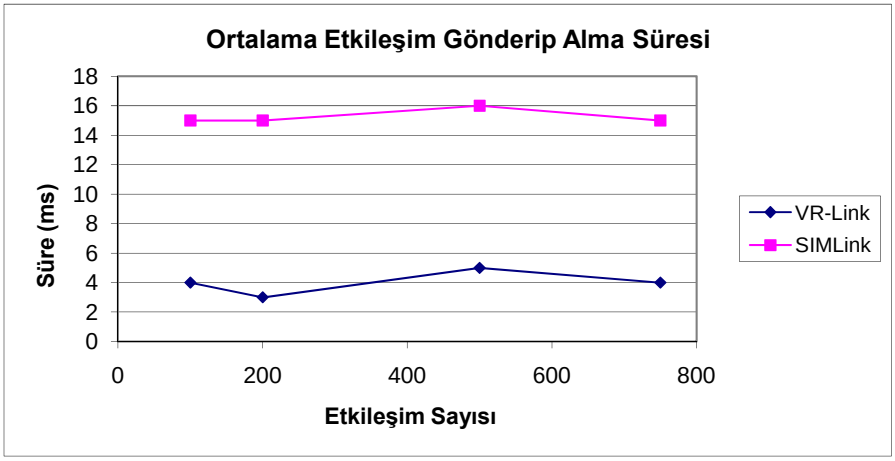
HLA Servisleri	**GENESIS**	**VR-Link**	**SIMLink**
Federasyon Yönetimi (Federation Management)	+	+	+
Gösterim Yönetimi (Declaration Management)	+	+	+
Nesne Yönetimi (Object Management)	+	+	+
Sahiplik Yönetimi (Ownership Management)	+	+	-
Zaman Yönetimi (Time Management)	+	+	-
Veri Dağıtım Yönetimi (Data Distribution Management)	+	+	-
Yönetim Nesne Modeli (Management Object Model)	+	+	+

4.10 Performans Karşılaştırma

SIMLink platformunun veri gönderim hızı ve ağ performansı açısından VR-Link altyapısı ile değerlendirilmesi bu kısımda ele alınmıştır. Bu amaçla iki farklı test gerçekleştirilmiştir. Testler, bu iki altyapı üzerine yazılan biri gönderici diğeri alıcı iki federe geliştirilerek yapılmıştır. Testlerde, veri boyutu değişken uzunluğa sahip bir etkileşim sınıfının, bu iki federe arasında gönderilip alınması sırasında bu platformların hız ve veri akış karşılaştırmaları yapılmıştır.

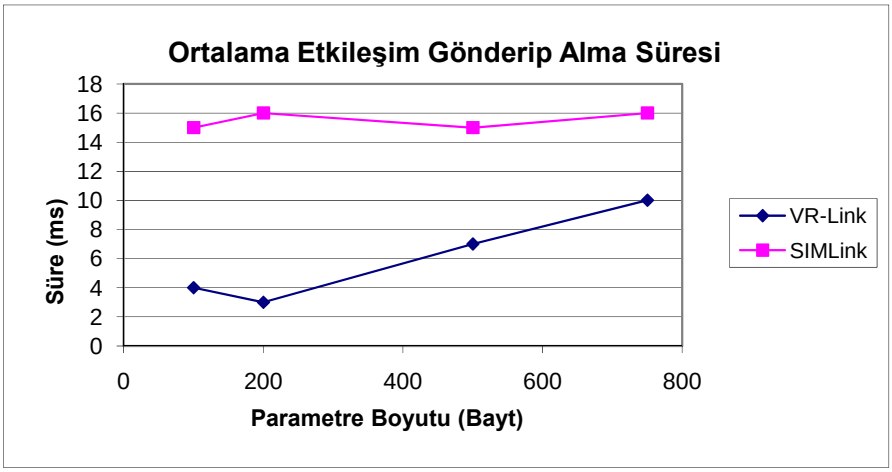
İlk olarak, belirli bir etkileşimin gönderici federeden alıcı federeye iletilmesi ve gönderici federeden geri döndürülmesi esnasında geçen sürenin etkileşim sayısına ve etkileşimin parametre veri boyutuna göre değişimi incelemiştir. Bu amaçla gönderici federe etkileşimi gönderdiği andaki zaman pulunu etkileşim içerisinde göndermekte ve aynı etkileşimin alıcı federeden kendisine döndüğü andaki zaman pulu ile karşılaştırarak etkileşimin ağ üzerindeki ortalama iletim süresini hesaplamaktadır.

Parametre boyu sabit (100 bayt) kalmak koşulu ile iletim süresinin etkileşim sayısına göre değişimi Şekil 4.12'de verilmiştir. Bu grafiğe göre geliştirilen platformun, VR-Link katmanına göre yaklaşık olarak 3 kat daha yavaş olduğu gözlenmiştir. Diğer taraftan her iki altyapı için, etkileşim sayısının arttırılması durumunda, bir etkileşimin ortalama gönderme süresinin hemen hemen aynı kaldığı gözlenmiştir.



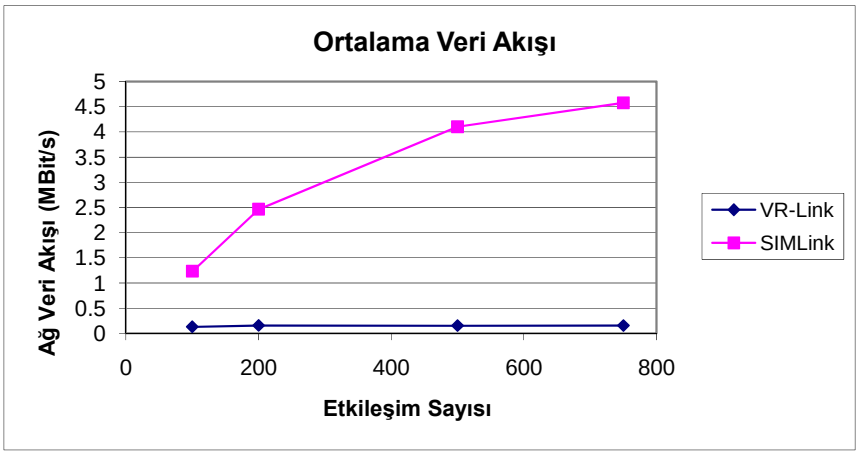
Şekil 4.12 : Etkileşim sayısı-süre grafiği.

İlk test, etkileşim sayısı sabit (100) tutularak etkileşimin parametre veri boyutu arttırılarak tekrar edildiğinde, veri gönderim hızındaki farkın aynı seyrettiği öte yandan geliştirilen uygulamanın artan veri boyutu için VR-Link uygulamasından daha kararlı seyrettiği gözlenmiştir. Alınan sonuçlar Şekil 4.13'de verilmiştir.



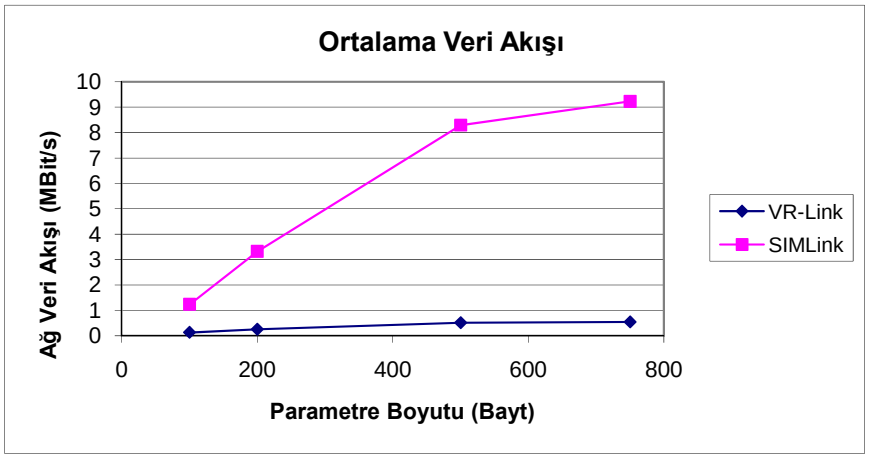
Şekil 4.13 : Parametre boyutu-süre grafiği.

İkinci test durumunda ise altyapılar ağ üzerindeki veri akışı performansları yönünden incelenmişlerdir. Bu amaçla ilk olarak, parametre boyutu sabit (100 bayt) bir etkileşimin belirli sayıda gönderimi ve tekrar alınması esnasında geçen süre ve etkileşim parametre boyutu kullanılarak bu işlem esnasında ağ üzerindeki toplam veri akışı incelenmiştir (Şekil 4.14). Buna göre, aynı sayıda etkileşim için VR-Link uygulaması tarafından ağ üzerindeki veri akışının geliştirilen altyapıya göre daha az olduğu gözlenmiştir. Veri akışındaki bu fark, geliştirilen altyapıda etkileşim sayısına bağlı olarak artışlar göstermiştir. Veri akışı arasındaki bu farkın, ilk test durumunda yaşanan gönderim zamanları arasındaki farka da etki ettiği değerlendirilmiştir. Gönderilen veri boyutu katmanlar tarafından çözülen/kodlanan veri boyutu ile aynı olduğu için bu boyutun, geliştirilen platformun bu fazla veri akışı esnasında verinin kodlanması ve çözülmesi için harcadığı zamanın oldukça artmasına neden olduğu değerlendirilmektedir. Özellikle ölçümler, etkileşimin kaynak federeden çıkışı ile tekrar kaynak federeye dönüşü arasında yapıldığından, katmanlar bu gönderim ve alım sırasında her etkileşim için ikişer defa kodlama ve çözme işlemi gerçekleştirmektedirler.



Şekil 4.14 : Etkileşim sayısı-veri akışı grafiği.

İkinci test durumunun sabit etkileşim sayısı (100) için değişen parametre boyutu ile ilişkisi Şekil 4.15'de sergilenmiştir. Önceki test durumlarından edinilen bilgiler doğrultusunda her birim veri için geliştirilen katman ile VR-Link katmanı arasında ağ üzerinde gönderilen veri boyutu olarak büyük farklar olduğu gözlenmiştir. Bu bilgilere dayanılarak, gönderilen parametre boyunun arttırılmasının, ağ üzerinde gönderilen veriler arasındaki artışın daha keskin bir şekilde artmasına neden olacağı beklenmektedir. Gerçekleştirilen test bu bilgilere uygun sonuçlar vermiş ve alınan sonuçlar Şekil 4.15'de gösterilmiştir.



Şekil 4.15 : Parametre boyutu –veri akışı grafiği.

4.11 Genişletilebilirlik-Tekrar Kullanılabilirlik Karşılaştırması

Geliştirilen platformların genişletilebilirliği ve benzer ihtiyaçlar için yeniden kullanılabilirliği önem taşımaktadır. Özellikle otomatik kod üretim metodu ile katmanları oluşturulmuş bu altyapılar tekrar kullanılabilirlik ve geliştirme kolaylığı açısından öne çıkmaktadırlar.

GENESIS platformunda kullanılan GenDL sayesinde modellenen yazılımın değişen ihtiyaçlar doğrultusunda güncellenerek yeniden kullanılması mümkündür. Ancak kullanılan dilin karmaşıklığı ve askeri amaçlı ihtiyaçlar doğrultusunda geliştirilmiş olması bu altyapının genişletilebilirliğini kısıtlamaktadır. Diğer taraftan mimarinin kod üretiminin şablonlar üzerine kurulu olması üretilen kodun şablon dosyaları değiştirilerek C++ dilinden farklı bir hedef dilde üretilmesini mümkün kılmaktadır.

VR-Link ürünü ise federe ve federasyonların standartlarla belirtilmiş olan nesne modelleri üzerine kurulu ve ticari bir ürün olduğundan tekrar kullanılabilirlik ve genişletilebilirlik açısından uygundur. Ancak bu ürün ile sunulan kod üretme aracı olan Code Generator uygulamasının yalnızca basit nesne modelleri için çalışabilir durumda olması, FOM eklentilerinde yeni sınıfların el ile kodlanarak kütüphaneye eklenmesini gerekli kılmaktadır. Diğer taraftan benzer nesne modelleri arasındaki geçişin bir uygulama programlama arayüzü (FOM Mapper) üzerinden sağlanması bu mimari üzerinde yazılmış uygulamaların minimum değişiklikle yeni amaçlar için kullanılabilmesini sağlamıştır. Ürün şu an itibari ile yalnızca C++ kütüphanelerini sunmaktadır.

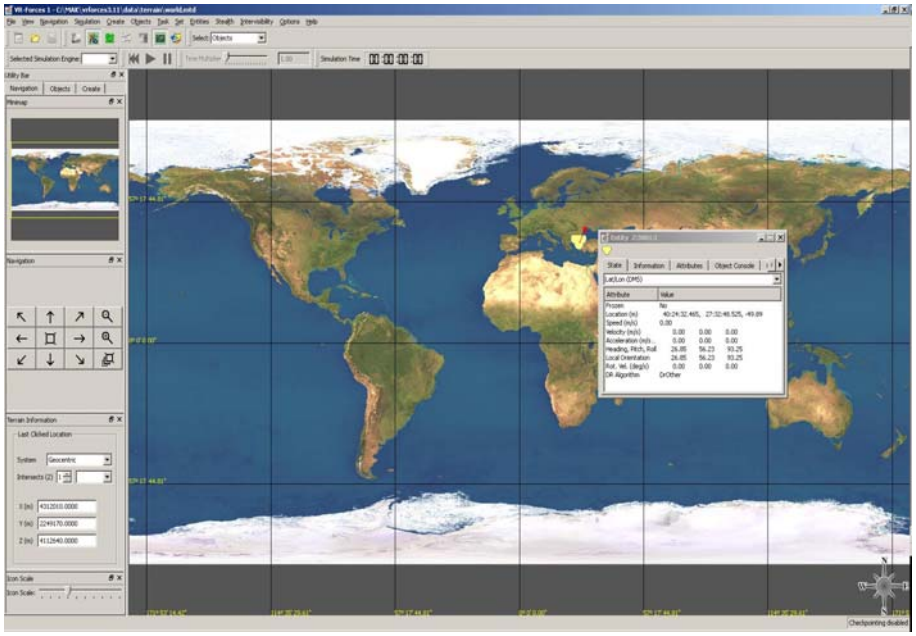
SIMLink ve bağlı olduğu katmanların nesne modeli üzerine kurulu olması ek bir girdi olmaksızın, federe tarafından kullanılacak olan kütüphane dosyalarının otomatik olarak üretilebilmesini sağlamaktadır. Üretilen kodun şablon temelli olması, şablon dosyalarının değiştirilmesi ile C++ dışında diğer hedef diller için de katmanların kullanılabilmesine imkan sağlamaktadır. Hazırlanan kütüphaneler nesne modeli içinde yer alan her tip veri için kod üretimini kullanıcıdan bağımsız olarak gerçekleştirebilmektedir. Her katmanda yer alan kütüphaneler için Windows işletim sistemi için kütüphane dosyaları üretilerek kullanıma sunulmuştur. Ayrıca FOMMapper kütüphanesi, yeni nesne ve etkileşim sınıfları için katmanların tekrar üretilmesine ihtiyaç olmaksızın mimarinin genişletilebilmesine olanak vermiştir.

4.12 Uyumluluk Karşılaştırması

Federelerin aynı federasyonda bulunabilmeleri aynı Federasyon Nesne Modelini kullanmaları ile mümkündür. Genel ihtiyaçlar doğrultusunda simülasyonların birlikte çalışabilirliğinin arttırılması amacıyla yazılmış RPR nesne modeli gibi modeller farklı altyapılar kullanılarak geliştirilmiş federelerin aynı federasyona dahil olarak veri alış verişinde bulunmalarını mümkün kılmıştır [12].

MAK firması tarafından sunulan VR-Link ticari ürünü bu nesne modelinin farklı versiyonları için kütüphaneler sunmuş olup, bu mimari üzerinde geliştirilen daha üst düzey diğer bir ticari simülasyon uygulaması olan VR-Forces uygulamasının da bu sayede standartlara uygun olması sağlanmıştır. Bu uygulama genel olarak bilgisayar tarafından üretilen platformların (CGF) simüle edilmesi amacıyla oluşturulan bir üründür. Ürün, benzetimini yaptığı bu platformların kullanıcı tarafından kontrolünü sağlamakla birlikte, federasyonda bu nesne modeline uygun olarak geliştirilmiş olan başka platformların da gösterimini yapabilmektedir.

Geliştirilen altyapı kullanılarak oluşturulacak federelerin de bu tip standartlara uygun olması bu tür ürünlerle haberleşebilmeleri açısından önem taşımaktadır. Üretilen kodun nesne modeli üzerinde oluşturulması bu uyumluluğun farklı federasyon ve simülasyon nesne modelleri için mümkün olmasını sağlamaktadır. Şekil 4.16'da uygulama üzerine geliştirilmiş olan bir federe tarafından simüle edilen bir helikopterin MAK firmasının VR-Forces ürünü üzerinde görsellenmesi sunulmuştur.



Şekil 4.16 : Nesne görüntüleme.

Bu amaçla VR-Forces ürününün uyumlu olarak geliştirildiği RPR 1.0 nesne modeli kullanılmış, bu nesne modeli için FOMLink katmanı otomatik olarak üretilmiş ve bu katmanı da kullanacak şekilde SIMLink katmanı üzerinde bir federe geliştirilerek, nesne modelinde yer alan helikopter nesnesinin oluşturulması ve özelliklerinin güncellenmesi bu federe tarafından sağlanmıştır. Bu nesnenin simülasyon ortamına gönderilmesi esnasında kullanılan kodlama mekanizması IEEE standartlarına uygun olarak gerçekleştirildiğinden, nesne VR-Forces ürünü tarafından görsellenebilmiş ve federenin yayınladığı özellikler bu ürün tarafından sergilenebilmiştir.

5. SONUÇ VE ÖNERİLER

Simülasyon sistemlerinin askeri alanlar başta olmak üzere, eğitim amaçlı gerçeklenmesi zor süreçlerin modellenmesi ve geliştirilen modellerin doğrulanması gibi birçok alanda kullanılmaya başlamasıyla birlikte bu sistemlerin ortak mimariler üzerine ve belirli standartlara uygun olarak geliştirilmesi giderek önem kazanmıştır. Bu bakımdan başta DIS, HLA ve TENA [13] gibi mimariler ve RPR-FOM gibi ortak nesne modelleri çeşitlenen bu sistemlerin geliştirilmesinin ve birlikte çalışabilmesinin kolaylaştırılması amacıyla ve ihtiyaçlara yönelik olarak sürekli geliştirilmektedir.

Simülasyon sistemi geliştiricilerinin bu mimari detaydan olabildiğince uzak tutularak, daha çok alan çalışmalarına yönelebilmesi ve yapılan çalışmaların tekrar kullanılabilir mimariler üzerinde yapılandırılması bu alanda yapılan çalışmaların önemli bir bölümünü oluşturmaktadır.

Çalışmada, bu amaca yönelik olarak kullanılabilecek bir altyapı mimarisi kod üretme teknikleri de uygulanarak oluşturulmuş, bu mimarinin benzer platformlarla mimari kullanım, kod üretim teknikleri, sunulan hizmetler, genişleyebilirlik, performans ve birlikte çalışabilirlik durumları değerlendirilmiş ve sonuçlara değinilmiştir.

Çalışma kapsamı bu amaçla gerekli HLA servislerinin sunulması ve örnek bir nesne modeli üzerinde C++ kodlarının Windows platformu için üretilmesi ile sınırlı tutulmuştur. Bundan sonraki çalışmalarda bu mimariye ileri HLA servisleri için hizmetlerin eklenmesi, kod üretiminin işletim sistemine uygun olarak kütüphaneler üretebilecek hale getirilmesi, üretilen kodun farklı hedef diller için de kullanılabilmesi, mimarinin dağıtık simülasyon sistemlerinin haberleşmesi amacıyla kullanılan diğer mimariler üzerinde (DIS, TENA) genişletilmesi ve performansının arttırılması konularının ele alınabileceği değerlendirilmektedir.

KAYNAKLAR

[1] **IEEE STD 1516,** 2000. Standard for Modeling and Simulation (M&S) High Level Architecture (HLA)-Framework and Rules, *IEEE,* New York.

[2] **IEEE STD 1516.1,** 2000. Standard for Modeling and Simulation (M&S) High Level Architecture (HLA)-Federate Interface Specification, *IEEE,* New York.

[3] **IEEE STD 1516.2,** 2000. Standard for Modeling and Simulation (M&S) High Level Architecture (HLA)-Object Model Template (OMT) Specification, *IEEE,* New York.

[4] **IEEE STD 1516.3,** 2003. Recommended Practice for High Level Architecture (HLA) Federation Development and Execution Process (FEDEP), *IEEE,* New York.

[5] **Belanger, W., Trainer, J., Ackerman, J., and Sanders, J. C.,** 1997. A Case Tool for Development of HLA Systems. *Simulation Interoperability Workshop*, Orlando, FL. USA.

[6] **Bourrely, J., Carle, P., Barat, M., and Lévy, F.,** 2005. GENESIS: an integrated platform for designing and developing HLA applications. *European Simulation Interoperability Workshop*, Toulouse, France.

[7] **Granowetter, L.,** 1999. Solving the FOM-Independence Problem. *Simulation Interoperability Workshop*, Orlando, FL. USA.

[8] **Baştürk, T., Avcıbaşı, Y. K., Destanoğlu, O., Eroğlu, Ö., Solter, H. G., Sevilgen F. E., et al.,** 2007. HLA Tabanlı Dağıtık Simülasyonlarda Federe Yönetim Altyapısı. *USMOS*, Ankara, Türkiye.

[9] **Govindaraju, M.,** 2007. XML Schemas Based Flexible Distributed Code Generation Framework. *IEEE International Conference on Web Services, ICWS.* Issue , 9-13 July, pp. 1212-1213.

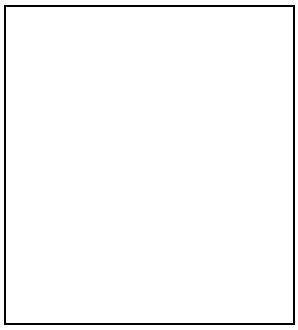
[10] **Herrington, J.,** 2003. Code Generation in Action, *Manning Publications, USA.*

[11] **Thomas, D., Fowler, C., and Hunt, A.,** 2005. Programming Ruby, The Pragmatic Programmer's Guide 2nd. Ed. *The Pragmatic Programmers*, USA.

[12] **Aldoğan D., Köksal S., Akdemir C., Taşdelen İ., and Dikenelli O.,** 2007. Farklı Simülasyon Sistemlerinin Entegrasyonu için Kontrol Mimarisi. *USMOS*, Ankara, Türkiye.

[13] **Poch, K.,** 2003. Test&Training Enabling Architecture (TENA). *NET3 Conference*, Orlando, FL. USA.

ÖZGEÇMİŞ



Ad Soyad: Cemil Akdemir

Doğum Yeri ve Tarihi: Karabük, 07.03.1981

Adres: TÜBİTAK Marmara Araştırma Merkezi Bilişim Teknolojileri Enstitüsü
Gebze, Kocaeli

Lisans Üniversite: İ.T.Ü. Bilgisayar Mühendisliği

Yayın Listesi:

▪ Aldoğan D., Köksal S., Akdemir C., Taşdelen İ., and Dikenelli O., 2007: Farklı Simülasyon Sistemlerinin Entegrasyonu için Kontrol Mimarisi. *İkinci Ulusal Savunma Uygulamaları Modelleme ve Simülasyon Konferansı*, Nisan 18-19, 2007 Ankara, Türkiye.