Developing JAUS Compliant Communication Infrastructures for Command & Control of Unmanned Systems through MDSD

İskender YAKIN
Faruk BELET
Ferhat KUTLU

Bilkent University, Computer Engineering Department, Ankara, Turkey
yakin,belet,fkutlu@cs.bilkent.edu.tr

Abstract— JAUS (Joint Architecture for Unmanned Systems) has enabled the development of Command & Control Systems (of unmanned vehicles) consisting of plug & play components. This property of JAUS has emerged from the ability that every component in the system can be addressed with a unique address called JAUS address and services provided by components can be registered with this address. Even if JAUS provides a flexible and extendable infrastructure for developing and unmanned systems, as the number of components increases in systems it becomes difficult to define the communication requirements among components without any design aid. In this project we aim to develop a graphical design tool for JAUS compliant systems through MDSD with the use of EMF (Eclipse Modeling Framework) and GMF (Graphical Modeling Framework) frameworks in Eclipse Ganymede 3.4.

Index Terms— JAUS, System, Subsystem, Node, Component and Instance, Abstract Syntax, Concrete Syntax.

1. INTRODUCTION

Unmanned systems reduce exposure of personnel to harmful environments, perform tasks not possible for humans, and provide cost effective solutions to repetitive tasks. As a result, a large number of unmanned system products are being introduced to the market. Furthermore, many of these systems are characterized as task dependent and non-interoperable. A standard open architecture is required to support the rapid and cost-effective development of unmanned systems. The Joint Architecture for Unmanned Systems (JAUS) is the architecture defined for use in the research, development and acquisition of Unmanned Systems. JAUS provides a description of the structure of JAUS based systems. It describes a component based system structure.

JAUS provides the necessary definitions to develop architecture to support the following objectives:

- Support all classes of unmanned systems
- Rapid technology insertion
- Interoperable operator control unit
- Interchangeable/interoperable payloads
- Interoperable unmanned systems

In order to achieve these objectives the interoperability among JAUS components plays a key role. Interoperability is the ability of two or more subsystems to exchange information and to use the information that has been exchanged [IEEE 90]. In order to achieve interoperability all components of a system must conform to predefined rules such as standardized addressing (JAUS Address) and messaging.

Even if JAUS provides a flexible and extendable infrastructure for developing unmanned systems, as the number of components increases in systems, it becomes difficult to define the communication parameters among components without any design aid. In this project we aim to develop a graphical design tool for JAUS compliant systems through MDSD with the use of EMF (Eclipse Modeling Framework) and GMF (Graphical Modeling Framework) frameworks in Eclipse Ganymede 3.4.

In Section 2, the domain analysis will be given by focusing on the domain description/context and domain lexicon. In Section 3 and Section 4, definition of metamodel based on MOF and using UML profiling, along with abstract syntax, concrete syntax, static semantics and example models will be described. In Section 5 and Section 7 Model to Model and Model to Text transformations will be given. Lessons learned about this work will be given in Section 7.

2. JAUS DOMAIN

2.1. Domain Description and Context

The purpose of JAUS is to support the acquisition of Unmanned Systems by providing a mechanism for reducing system life-cycle costs. This is accomplished by providing a framework for technology reuse/insertion. JAUS defines a set of reusable “components” and their interfaces. These reusable components not only reduce the maintenance costs of a system, but also dramatically reduce the development costs of any follow-on system(s). Reuse allows a component developed for one Unmanned System to be readily ported to another Unmanned System or to be easily replaced when technological advances.

Technology insertion is achievable when the architecture is designed to be both modular and scalable. Components that

are deemed necessary for the mission of the Unmanned System may be inserted simply by bundling.

JAUS defines components for all classifications of Unmanned Systems from remote control toward autonomous, regardless of application. As a particular system evolves, the architecture is already in place to support more advanced capabilities. In JAUS language, a number of terms are used to delineate position within the overall hierarchy of the system and must therefore, be well understood. These terms describe the different levels of the architecture and define the required internal hierarchical sub-grouping:

- System: A system is a logical grouping of subsystems. The system definition provides a functional grouping for the full robotic or unmanned capability. This grouping includes all human interface subsystems and unmanned subsystems common with robotic and unmanned applications.
- Subsystem: A subsystem performs one or more unmanned system functions as a single localized entity within the framework of the System. A subsystem shall provide one or more communication command and control capabilities. A mobile subsystem shall execute mobility commands as a single unit and retain a defined center of gravity relative to all articulations and payloads.
- Node: A JAUS Node defines a distinct processing capability within a subsystem. A node retains a set of coherent functions and shall provide a node manager component to manage the flows and controls of JAUS message traffic.
- Component: A component provides a unique functional capability for the unmanned system. JAUS messages are defined with respect to these capabilities so that context in command and control is provided. A JAUS component resides wholly within a JAUS Node.
- Instance: Duplication and redundancy of JAUS Components are provided by Component Instances. All Components are uniquely addressable using Subsystem, Node, Component and Instance Identifiers.
- Message: A JAUS message is comprised of the message header and associated data fields.

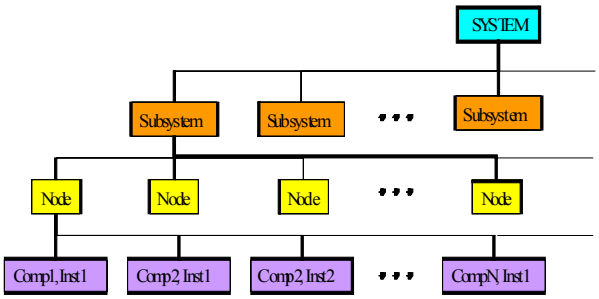

Figure 1: JAUS System Topology

A graphical representation of how each element fits into the JAUS system topology is shown in Figure 1.

2.2. DSL Grammar

Since the JAUS communication infrastructure consists of hierarchical elements it is easy to define this infrastructure by a grammar in BNF. The following grammar in Figure 2 describes communication infrastructure.

```
<System> ::= <SystemEntity> <SubsystemList>

<SubsystemList>::=<Subsystem> | <Subsystem>
                  <SubsystemList>

<Subsystem> ::= <SystemEntity> <NodeList>

<NodeList > ::= <Node > | <Node> <NodeList>

<Node>::=<SystemEntity><IPAdsress><ComponentList>

<IPAddress>::= <Number>.<Number>. <Number> .
                           <Number>

<ComponentList>::=<Component>|<Component>
                   <ComponentList>

<Component> ::= <SystemEntity> <InstanceList>

<InstanceList> ::= <Instance> | <Instance>
<InstanceList>

<Instance>::=<SystemEntity><JAUSAddress>
              <ConnectionList> <ConnectionList>

<JAUSAddress> ::= <Number> . <Number> . <Number>.
                   <Number>

<ConnectionList>::=<CommLink>|<CommLink>
                   <ConnectionList>
<CommLink> ::= <Instance> <Instance> <Port>

<SystemEntity> ::= <name> <ID>
```

Figure 2: JAUS Communication infrastructure grammar.

3. DEFINITION OF METAMODEL BASED ON MOF

3.1. Abstract Syntax

Metamodel of our system in M2 level is illustrated in Figure 2. In the metamodel all system entities which are system, subsystem, node, component and instance are derived from a class called SystemEntity. In this model System is the highest level object in the system and it contains subsystems. As the level of containment decreases subsystems contain nodes, nodes contain components and at the lowest level instances are contained by components. Only instances communicate with each other and so only instances have unique JAUS addresses. Every instance has source and target connections meaning that every connection originating from an instance is added to source connections of that node and the same connection is added to the target connections of the other node which is targeted by that connection.

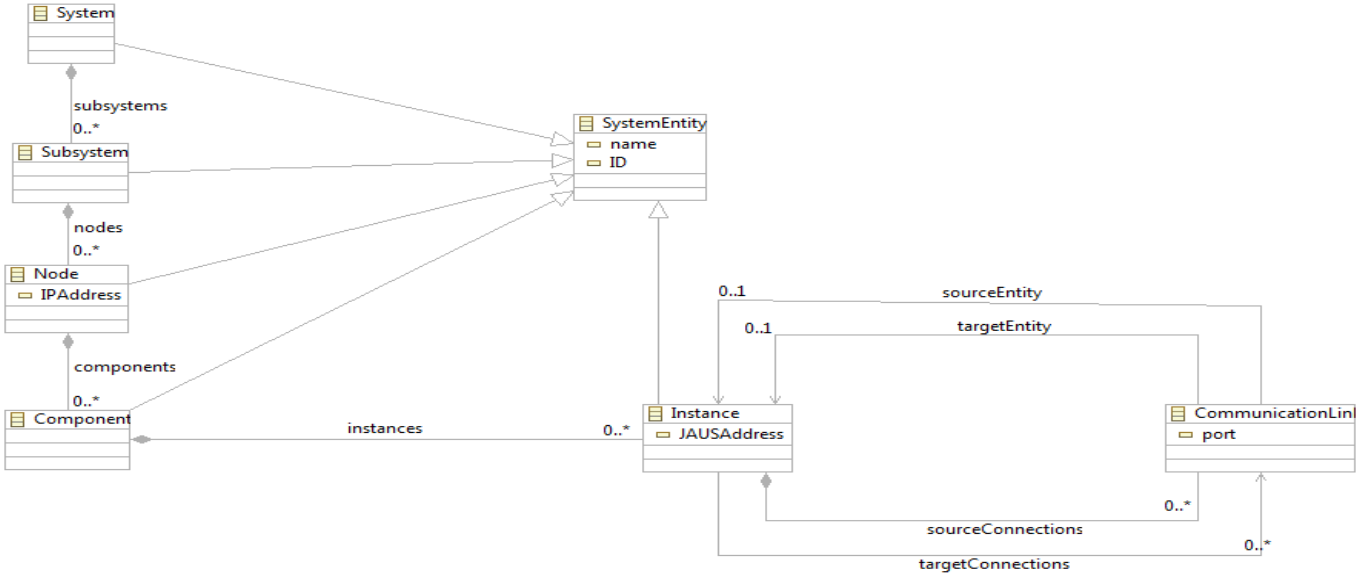

Figure 3: JAUS System Metamodel

Metamodel at M2 level is formed from Eclipse EMF/GMF metamodel at M3 level given in Figure 3 and 4. With respect to this model at M3 level all system entities, system, subsystem, node, component instance and are derived from Node and communication links are derived from Edge.

3.2. Concrete Syntax

Visual definition of system entities if defined in the Graphical Definition Model of Eclipse GMF Project as illustrated in Figure 4.

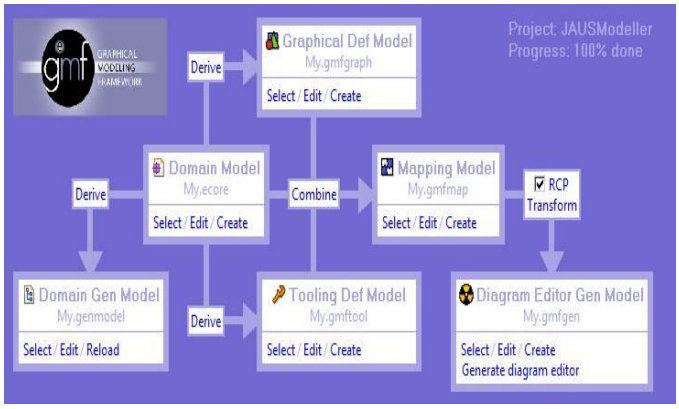

Figure 4: Eclipse GMF Project Development Phases

The graphical definition of our system as a Graphical Definition Model is as given in Figure 5. As can be seen in the figure the concrete syntax of metamodel elements are defined graphically in the Graphical Definition Model. For example, the graphical definition of 'Subsystem' in the metamodel is given under the "Figure gallery Default". 'Subsystem Figure' corresponding to Subsystem is displayed in our editor as a rectangular figure, and in the figure we display the name and ID of this entity as can be seen in the Graphical Definition Model.

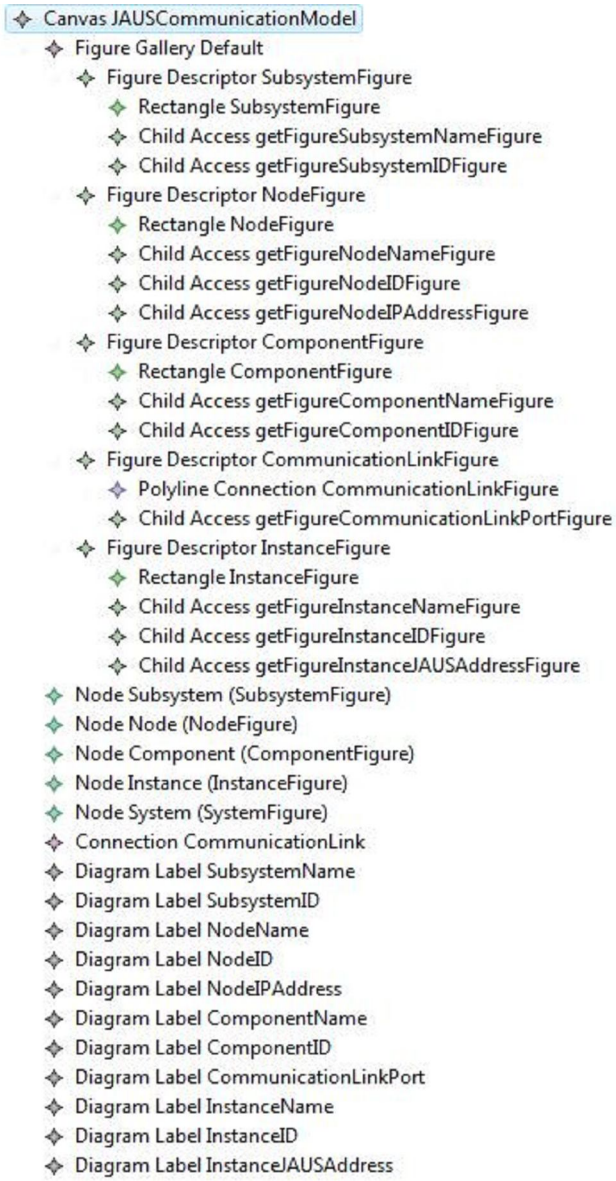

Figure 5: Graphical Definition Model of GMF

3.3. Static Sematics

JAUS communication metamodel contains only ordinary system checks because it has a hierarchical structure and well isolated parts. The OCL constraints used in the concrete syntax are as follows.

IP Address Control: Control of the IP addresses in Nodes to sustain each Node has a unique IP Address.

context Subsystem inv:

```
Node.allInstances()->forAll(n1|Node.allInstances()
    ->forAll(n2|n1.IPAddress<>n2.IPAddress))
```

JAUSAddress Control: Control of the JAUS addresses in Instances to sustain each Instance has a unique JAUS Address.

context Component inv:

```
Instance.allInstances()->forAll(j1|Instance.allInstances()
    ->forAll(j2|j1.JAUSAddress<>j2.JAUSAddress))
```

Port Control: Control of the ports in Communication Links to sustain each Communication Link has a unique port.

context Instance inv:

```
CommunicationLink.allInstances()
  ->forAll(p1|CommunicationLink.allInstances()
      ->forAll(p2|p1.port<>p2.port))
```

Source-Target Inequality Control: Control of the connections in an Instance to sustain sources and targets to be different Instances.

context Instance inv:

```
CommunicationLink.allInstances()
      ->forAll(c|c.sourceEntity<>c.targetEntity)
```

3.4. Example Model

The following model in Figure 6 is derived from our metamodel is at M1. It is at M1 since the auto generated code from this model is at M0. The system defined by this model consists of three subsystems, each of which contains a single node and each node contains a single component with a single instance.

Figure 6: Example JAUS Communication Model

The model shows that Commander only communicates with the operator and the operator communicates with both the observer and the interceptor

4. Definition of Metamodel using UML Profiling

4.1. Abstract Syntax

A profile in UML enables generic extension mechanism for customizing UML models for some domains. Profiles are defined using stereotypes, tagged values and some constraints that are applied to specific model elements in UML metaclasses, such as Class, Attributes and Operations.

The profile is not necessary to make UML applicable to this domain. As a lightweight extension mechanism, Class metaclass is extended by domain stereotypes such as System, subsystem, node, component etc. In addition to this, Association and Aggregation metaclasses are used to extend existing domain specific association and aggregation between the domain model objects (see Figure 7).

4.2. Example Model

An example model in Figure 8.

5. Model to Model Transformation

In this project, model to model transformation is based on two steps. Firstly, transformation from Ecore Model to Class Schema Model is handled. After transformation, as second step Class Schema Model is transformed to Relational Database Model. ATL eclipse plug-in is used to transform between these models.

The Class metamodel (see Figure 9) consists of classes having a name that inherit from the abstract class NamedElts. Class "Class" is the principal class, which consist of a set of attributes of the type "Attribute" and has super references pointing to super-classes for modeling inheritance trees. Primitive data types are modeled by the class "DataType" which inherits from Classifier. "Class" also inherits from Classifier. Classifier is utilized for declaration of "Attribute" type. Attributes can be multi-valued, which has an important impact on the transformation.

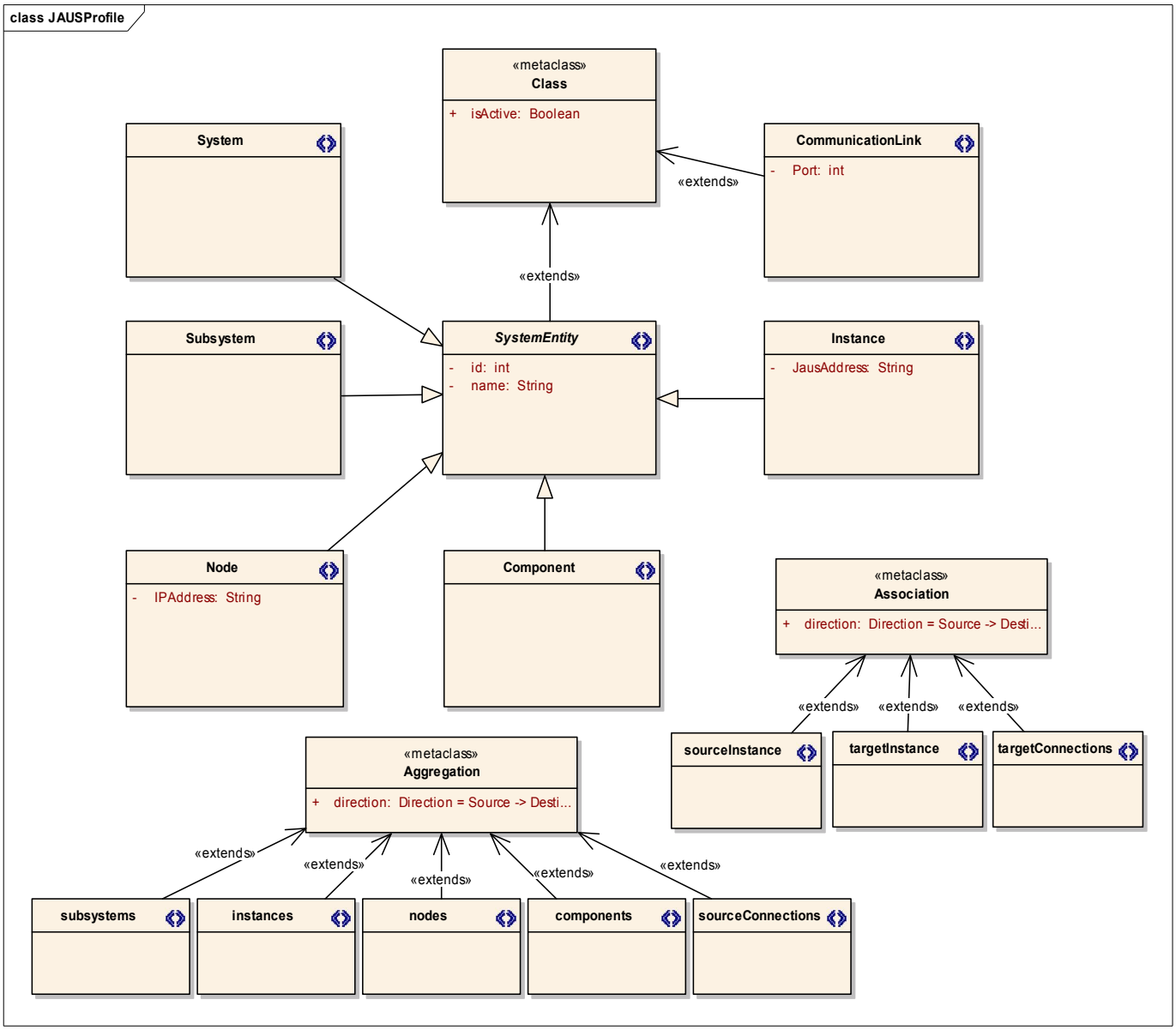

Figure 7: UML Profiling for JAUS System

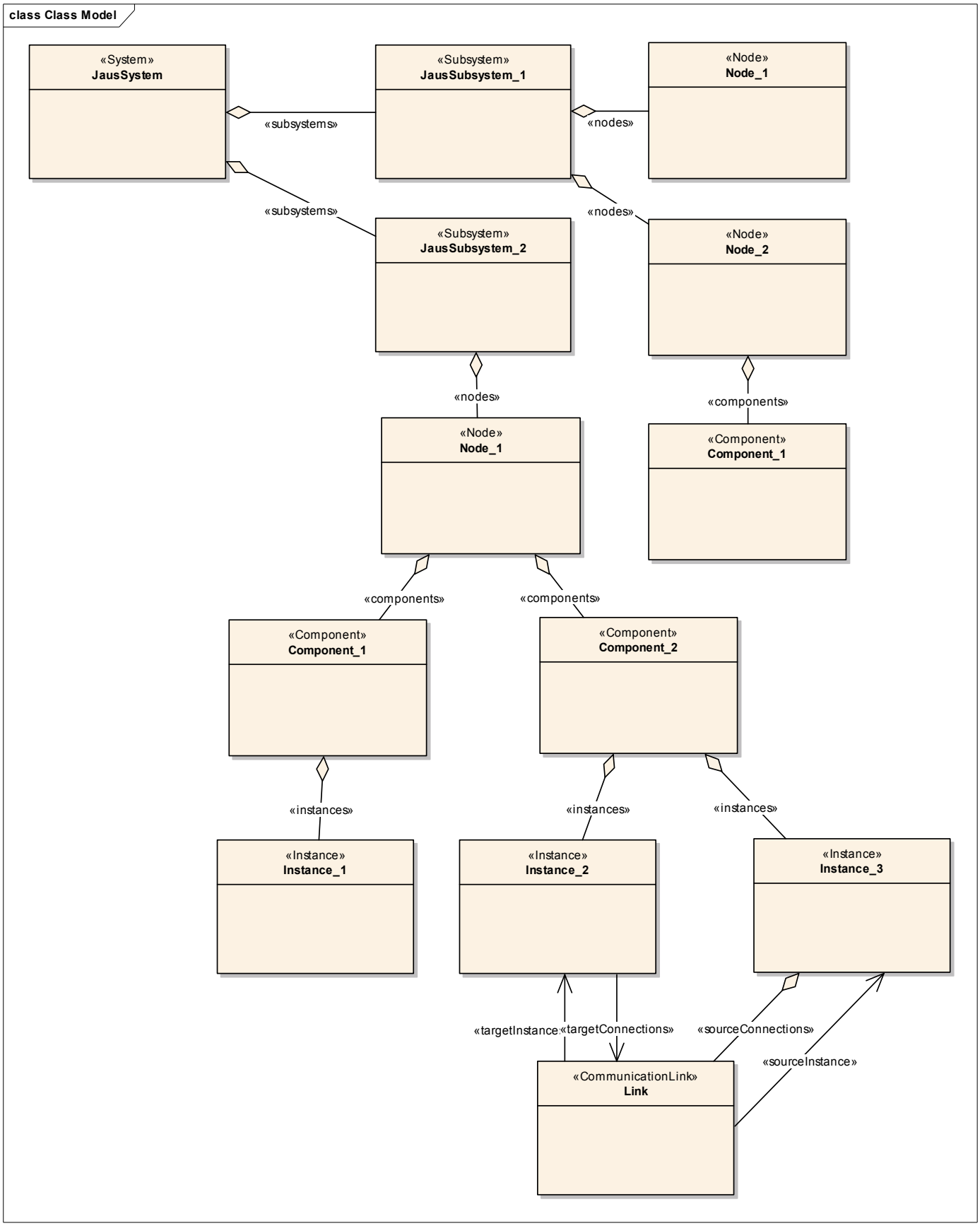

Figure 8: Example Model

The Relational metamodel (see Figure 10) has classes having a name which they inherit from the abstract class "Named". "Columns" is the principal class table and has a reference to its keys. "Column" class has the references owner and keyOf pointing to the Table it belongs to and of which it is part of the key. It also reference to "Type"

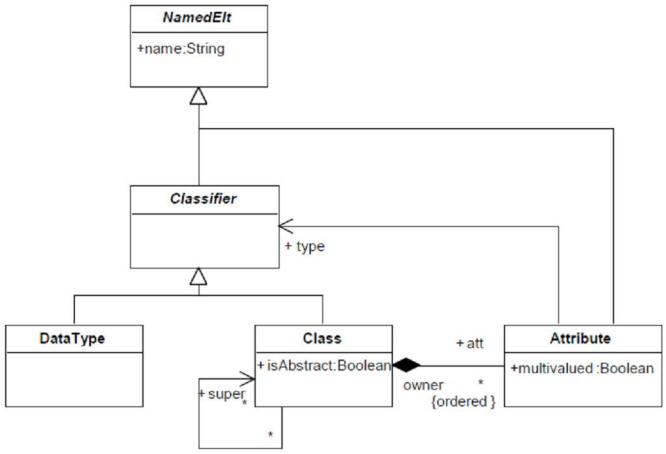

Figure 9: Class Meta Model

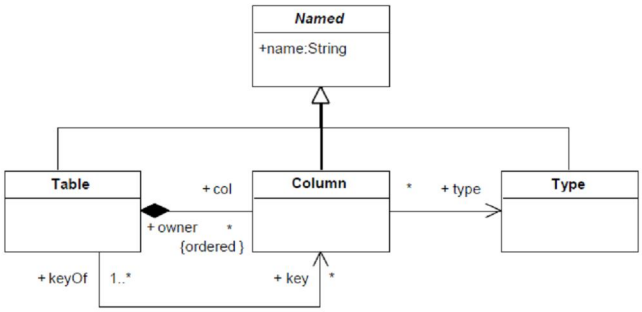

Figure 10: Relational Meta Model

ATL enables us to define target and source metamodels. In addition, source ecore is used to get a transformed target output that conformes to the target metamodel. Skipping transformation of the Ecore model to Class Schema Model details, we will introduce the part telling how to convert a Class Schema Model to Relational Meta model.

Next step of the conversion is to define the rule specification for transformation between models. These are the rules to transform a class model to a relational model:

- For each Class instance, a Table instance has to be created.
- For each DataType instance, a Type instance has to be created (see Figure 11).
- For each single-valued Attribute instance of the type DataType, a Column instance has to be created.
- For each multi-valued Attribute instance of the type DataType, a Table instance has to be created.
- For each single-valued Attribute of the type Class, a new Column has to be created.
- For each multi-valued Attribute of the type Class, a new Table has to be created.

In addition to these rules, for the simplification's sake inheritance and isAbstract are not taken into consideration.

```
rule DataType2Type {
        from
                dt : Class!DataType
        to
                out : Relational!Type (
                        name <- dt.name
                )
}
```

Figure 11 Example Rule

6. MODEL TO TEXT TRANSFORMATION

As mentioned in Section 2, JAUS communication infrastructure requires IP and JAUS addresses of JAUS components to be specified as an interface at each component if there is a communication link between two components.

This introduces the problem of interface maintenance whenever a JAUS or IP address changes. In order to overcome this problem .java files implementing these interfaces can be generated automatically. This reduces time spent for maintenance and also reduces the time for testing the modified system. Since we have the communication parameters in our model, changing the parameters at the model and generating the code dependent on this parameters is possible through model to text (java code) transformation.

For model to text (M2T) transformation we use Eclipse JET (Java Emitter Templates) plugin. In order generate code first the template for the java file should be coded. A JET template is a text file with a file name that end with "jet", .javajet implies that the template generates a .java file. A JET template includes both the static part of the code such as,

```
// Commander
        new UDPInterface(inetAddress, patameter, false, true)
```

and dynamic parameters that should be put into the code. For this example the code segment 'parameter' is the data that we are looking for in our model. When this parameter is passed from EMF .ecore to the template we get the following auto generated code line,

```
new UDPInterface(inetAddress, 1001, false, true)
```

As can be seen in Figure 6, 1001 corresponds to the port number between the operator and the commander of the system. Two code segments from the auto generated code in "OperatorNode.java" are given below. As can be seen, these code segments include the static code from JET template (code not marked with red), and data as parameters from the model given in Figure 6.

```
// Commander
DefaultCommunicator.getInstance().addInterface(
        new UDPInterface(inetAddress, 10001, false, true));

// Commander
ActiveRoute route = new ActiveRoute();
```

```
route.setInterfaceIndex(1);
route.setDestination(new JAUSAddress(1, 1, 255, 255));
route.setGateway(new JAUSAddress(1, 1, 32, 1));
route.setIfGateway("15.30.5.1:10001");
DefaultCommunicator.getInstance().addActiveRoute(route);
```

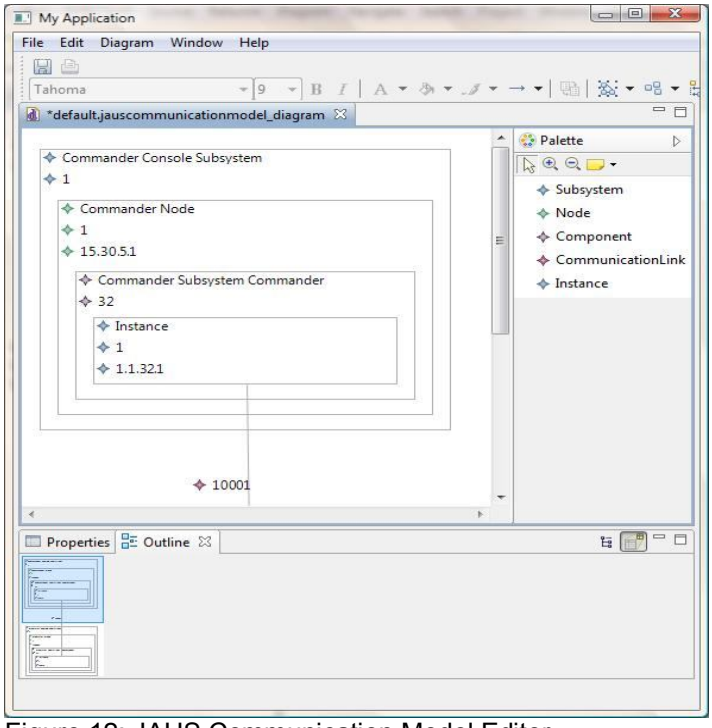

Figure 12: JAUS Communication Model Editor

Another model to text transformation example in this work is our JAUS Communication Editor illustrated in Figure 12. By using EMF and GMF this editor was generated from the set of models given in Figure 4. The whole editor application consists of around 35.000 lines of code and it is auto generated by GMF templates.

7. LESSONS LEARNED AND CONCLUSIONS

Determining the scope of a metamodel focusing the related concepts and whether it meets our system is actually a though part of Model Driven Software Architecture. Software development is inherently difficult because of the increasing complexity and frequent change that occurs during its lifecycle. However, high level of abstraction is required to handle all these drawbacks in software development.

We learned how to make a high level abstraction by using Ecore, UML profiling, and language grammar in a specific domain which is JAUS.

We experienced that meta-model definition takes a considerable amount of time. Implementing a given architecture is decreases the design time but requires the architecture to be understood in detail. 100% auto generated code with Eclipse EMF/GMF is achieved. However, it takes a great amount of time to define the models and mapping among these models. GMF has some problems with creating the Mapping Model. We noticed that using set/create transactions in EMF is required to set a value of the attribute of to create an instance of an entity manually. Changing requirements are easily reflected to the models. BNF Grammar helps to understand the system and the structure of system entities.

Model to Model transformations is a valuable experience to convert a domain model into another model that conforms to their metamodels. In our case, Relational Metamodel for storage case of our system is seen as a good practice. By this way, we haven't spent time to plan Relational Database Schema for the project as a separate focus. It is enough to define transformation rules. However, some possible rules are skipped for simplicity to achieve basic transformation.

By using model-to-text transformations code including the communication parameters of the system can be auto generated. This reduces time spent for maintenance and also reduces the time for testing the modified system. Since we have the communication parameters in our model, changing the parameters at the model and generating the code dependent on this parameters is possible through model to text (java code) transformation.

ACKNOWLEDGMENT

The authors would like to thank Dr. Bedir Tekinerdoğan for his magnificent guidance and leadership.

REFERENCES

[1] Hui Huang, Autonomy Levels for Unmanned Systems (ALFUS) Framework, JAUS WG Meeting August 26, 2004. Pittsburgh, PA.

[2] C4ISR Handbook for Integrated Planning (CHIP), DoD Integrated C4I Architectures Division, April 1998

[3] C4ISR Interoperability Working Group, Department of Defense. Levels of Information Systems Interoperability (LISI). Washington, D.C., 1998

[4] NATO C3 Technical Architecture Volume 2 Architectural Descriptions and Models Version 7.0, 15 December 2005

[5] JAUS, Domain Model, Volume 1. Version 3.2., 10 March 2005.

[6] JAUS, Reference Architecture Specification, Volume 2, Part 1, Architecture Framework