1 July 2015

Design Documentation

SysML to AADL Translator

Developer’s Manual

Version 1.0

Prepared for DARPA STO

System of Systems Integration Technology and Experimentation

SoSITE

Contract FA8650-14-C-7437

|  |  |
| --- | --- |
| **Technical Point of Contact:**  **Dr. Steven P. Miller**  Rockwell Collins, Inc.  400 Collins Road NE  Cedar Rapids, IA 52498  Telephone: (319) 295-2055 | **Business Point of Contact:**  **Ms. Kelsey Rwayitare**  Rockwell Collins, Inc.  400 Collins Rd. NE, MS 121-200  Cedar Rapids, IA 52498  Telephone: (319) 263-3101  [kelsey.rwayitare@rockwellcollins.com](mailto:kelsey.rwayitare@rockwellcollins.com) |

The views, opinions, and/or findings expressed are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.



**Table of Contents**

[1 Introduction 1](#_Toc421017866)

[2 The Translator Architecture 2](#_Toc421017867)

[2.1 Creating the Ecore Model 3](#_Toc421017868)

[2.2 Generating the XMI Parser 6](#_Toc421017869)

[2.3 Generating AADL 7](#_Toc421017870)

[2.4 Translation of State Machines 9](#_Toc421017871)

**List of Figures**

[Figure 1 - SysML to AADL Translator Overview 3](#_Toc421017872)

[Figure 2 – The Ecore Meta Model and XMI File 4](#_Toc421017873)

[Figure 3 - Generating the XMI Parser 6](#_Toc421017874)

[Figure 4 - Ecore Model and the XMI Parser 7](#_Toc421017875)

[Figure 5 - Translating EMF model to AADL Objects 8](#_Toc421017876)

[Figure 6 – Example State Machine 9](#_Toc421017877)

[Figure 7 - States in AGREE 10](#_Toc421017878)

[Figure 8 - Guaranteeing Only One Child State Active 10](#_Toc421017879)

[Figure 9 – Local Variables 11](#_Toc421017880)

[Figure 10 – Specification of the Initial State 11](#_Toc421017881)

[Figure 11 – Specification of Transition T4 11](#_Toc421017882)

[Figure 12 – Composite Transition into Meta-States 12](#_Toc421017883)

[Figure 13 – During Transitions 12](#_Toc421017884)

[Figure 14 - No Change Guarantees 13](#_Toc421017885)

|  |  |  |  |
| --- | --- | --- | --- |
| **Release History** | | | |
| **Date** | **Version** | **Author** | **Description** |
| July 1, 2015 | 1.0 | Siddhartha Bhattarcharyya | Initial Release |
|  |  |  |  |
|  |  |  |  |

# Introduction

The SysML to AADL translator translates system architectural models specified in a subset of the Systems Modeling Language (SysML) to models specified in a subset of the Architectural Analysis and Description Language (AADL). SysML is an open standard published by the Object Management Group (OMG) for the specification of system architectures. AADL is an open standard published by the Society of Automotive Engineers (SAE) for the specification of system architectures for embedded systems.

The SysML models are created using the Sparx Systems Enterprise Architect® (EA) tool and then exported as an Extensible Markup Language (XML) Metadata Interchange (XMI) file using the export option provided by EA. The translator currently supports the 2.1 version of XMI. A SysML profile for AADL extends SysML with constructs commonly used in AADL. These include AADL component categories such as system, process, thread, data, device, processor, bus and memory and AADL features such as port, data access, and bus access. These constructs are provided on an AADL toolbar palette in Enterprise Architect to simplify the development of SysML models that can be translated to AADL.

The SysML translator only translates a subset of the SysML constructs. A mapping of SysML constructs and the AADL constructs to which they translate is shown in Table 1.

The translator is invoked from within the OSATE (Open Source Architectural Tool Environment). OSATE is an open-source Eclipse-based framework for constructing and analyzing AADL models. The translator is written in the Java programming language and is packaged as plug-ins for the Eclipse development environment containing Java source code, executable byte code, and supporting XML files. It can be installed using the standard Eclipse facilities for installing new software. Once installed, new menu items are provided to the OSATE user to import a XMI-based XML model generated by the SysML tool to generate an AADL model.

Detailed instructions for installing and using the translator are provided in the User’s Manual. This document provides information for updating and maintaining the translator. The translator would likely need to be updated for versions of EA other than EA, or in the event that SysML constructs not currently supported for translation to AADL are required for a particular program or task.

Table 1 - SysML to AADL Objects

|  |  |
| --- | --- |
| **SysML Structure** | **AADL Object** |
| Block | ComponentType |
| Block as a Client in Realization Relation | ComponentImplementation |
| Flowports | Ports |
| Binding Connector | Connection |
| SysML part (block property) | Subcomponent |
| Multiplicity of parts or ports | Arrays of subcomponents or ports |
| State | Boolean variable in an AGREE annex |
| Transition | Boolean variable in AGREE |
| Transition Guard | Boolean expression (in transition) |
| Transition Action | Guarantee in a ComponentType  Assertion in a ComponentImplementation |
| Tagged Value | AADL subcomponent or connection property |

# The Translator Architecture

This section is an overview of the SysML to AADL translator architecture a high-level representation is shown in Figure 1. Initially, a SysML model is developed in Sparx EA and exported as an XMI 2.1 file. To create the translator, an Ecore model is first developed in Eclipse describing the meta-structure of the XMI file. This Ecore model can them be used to generate a parser that will read the XMI file and create an Eclipse Modeling Framework (EMF) model within Eclipse.

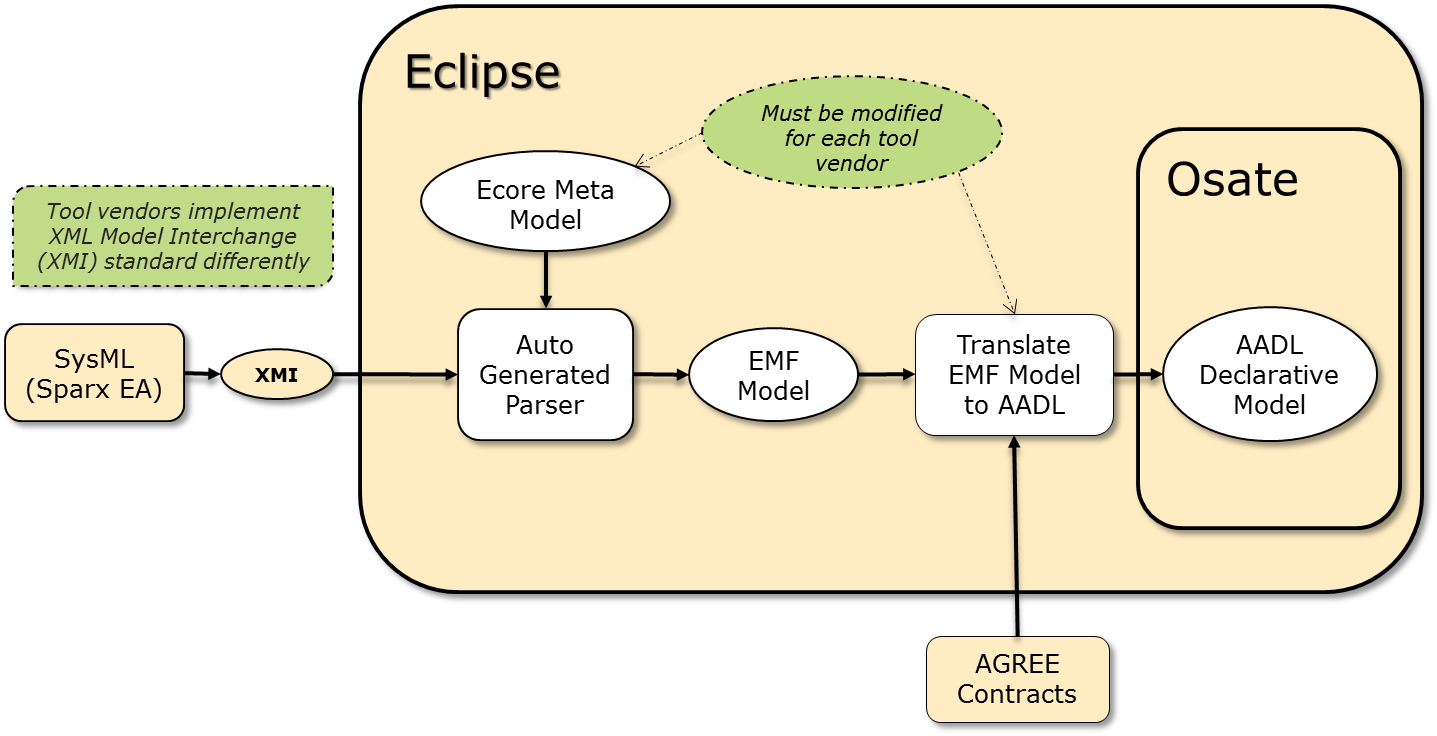


Figure 1 - SysML to AADL Translator Overview

The translator then traverses the EMF model and generates the AADL model by making calls to the Application Program Interface (API) provided by OSATE.

## Creating the Ecore Model

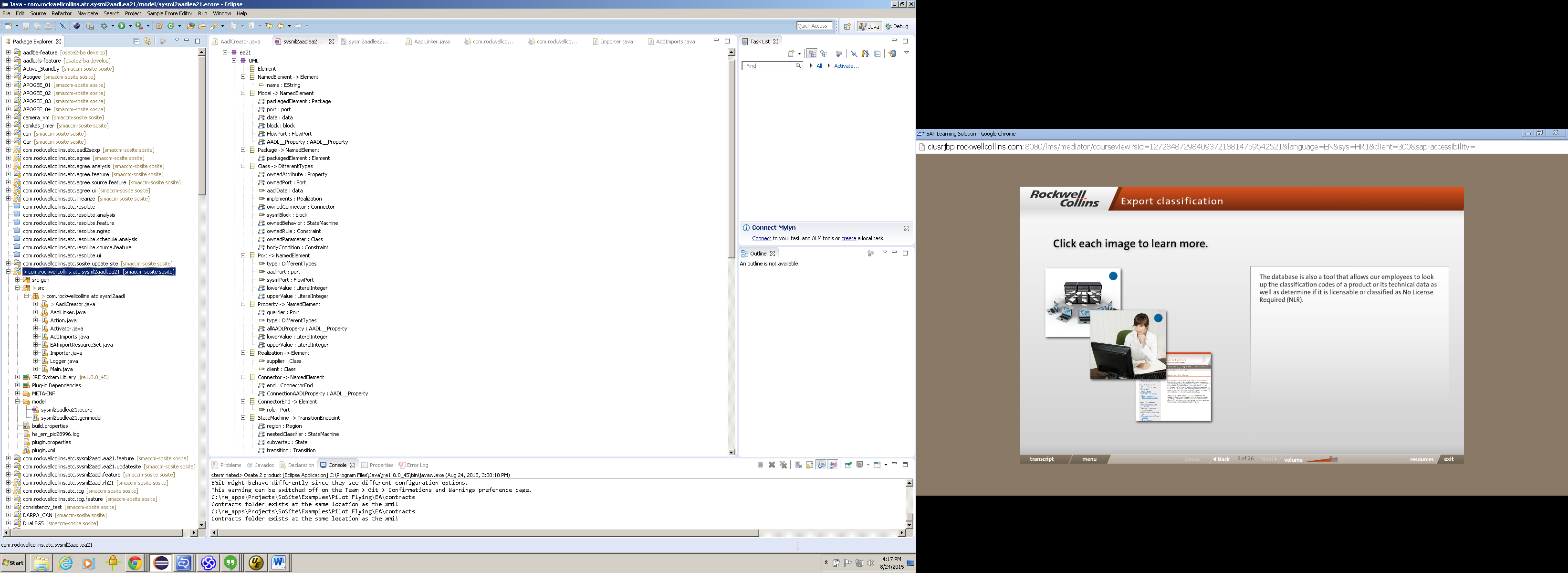
This section describes how to develop the Ecore model based on an understanding of the XMI structure.[[1]](#footnote-2) A tutorial on developing an Ecore model is available at <http://eclipsesource.com/blogs/tutorials/emf-tutorial/>. A portion of the Ecore model for EA is shown in Figure 2 .

The entire model is placed within the *ea21* package (i.e. the Eclipse Java package “com.rockwell.atc.sysml2aadl.ea21”). Examining the XMI file we find that the XMI model resides within the tag **<uml:Model>** with type **uml:Model**. So in Ecore we create a nested package named *UML* as shown in Figure 2.

The XMI model elements typically have attributes associated with them such as name, body etc. As name is the most widely used attribute it is convenient to create a *NamedElement* EClass model element with an attribute *name* ETyped to EString. To accommodate model elements that do not have a name we also create an *Element* EClass model element.

XMI File Exported from SysML

Ecore Meta Model



<uml:Model xmi:type="uml:Model" name="EA\_Model"

<packagedElement xmi:type="uml:Package" xmi:id="EA

<packagedElement xmi:type="uml:Package"

<packagedElement xmi:type="uml:Class”

<ownedPort xmi:type="uml:Port"

<type xmi:idref="EAID\_6

</ownedPort>

</packagedElement>

<packagedElement xmi:type="uml:Class"

<ownedPort xmi:type="uml:Port"

<type xmi:idref="EAID\_7

</ownedPort>

<ownedConnector xmi:type="uml:

<end xmi:type="uml:Con

<end xmi:type="uml:Co

</ownedConnector>

</packagedElement>

</packagedElement>

</packagedElement>

<AADL\_Elements:system base\_Class="EAID\_6420855F\_DF

<AADL\_Elements:data base\_Class="EAID\_F9EE3B1F\_D2B9

<AADL\_Elements:port base\_Port="EAID\_1E84D739\_552

<SysML:block base\_Class="EAID\_C9A76D6F\_4F2D\_47c3\_

</uml:Model>

Figure 2 – The Ecore Meta Model and XMI File

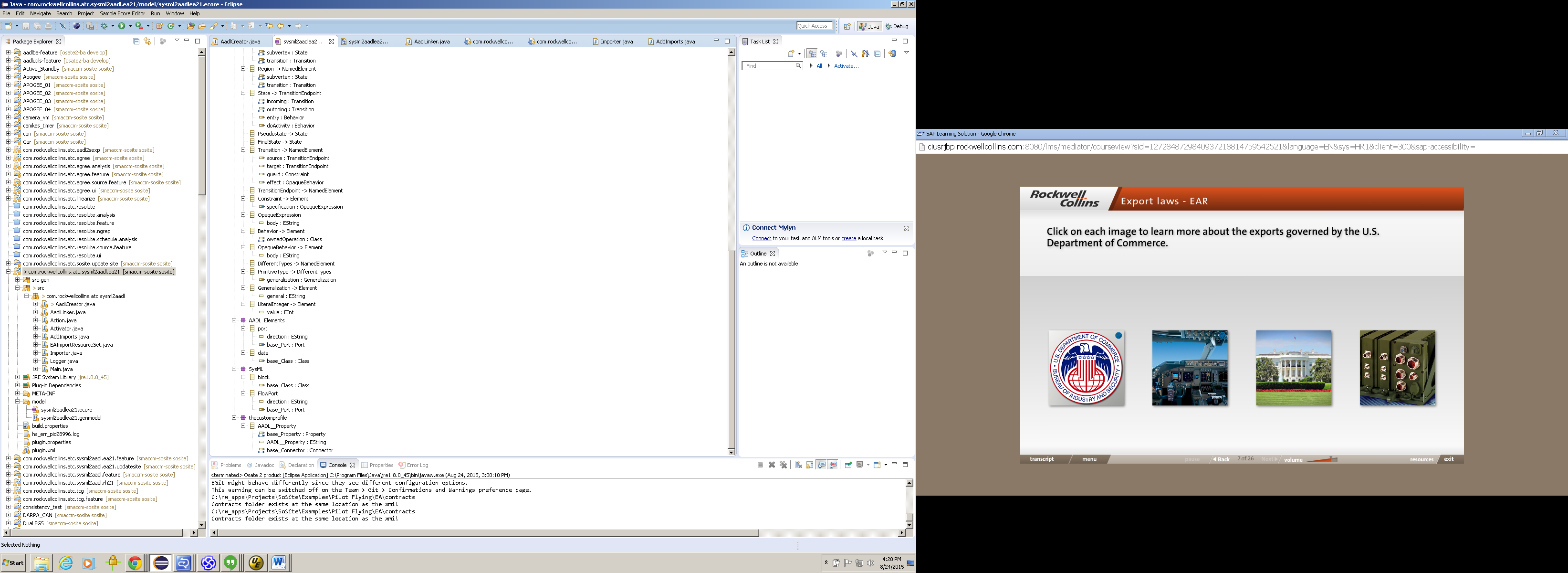
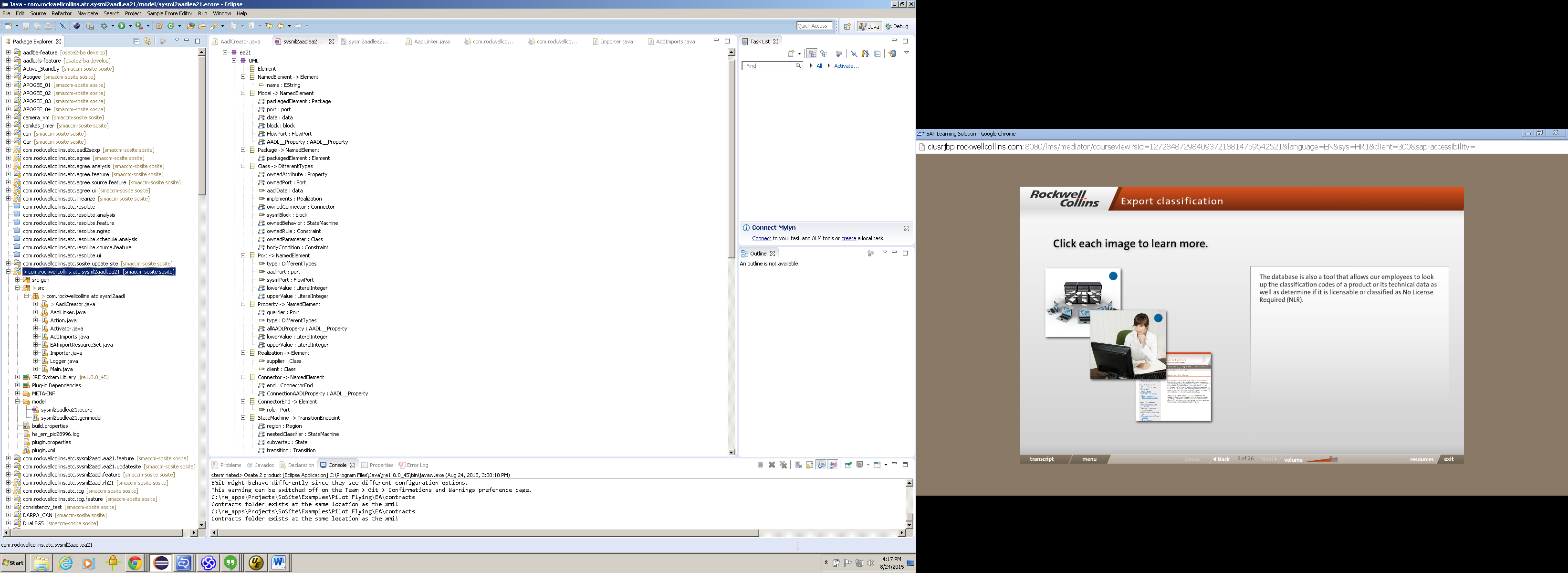


Figure 3: Complete Ecore Model

Further examination of the XMI model shows that the tag < **packagedElement…**> with type **uml:package** is embedded within the start and end of **<uml:Model>**. We create an EReference within *Model* and name it *packagedElement* just as within the tag. This new EReference thenneeds to beETyped to Package. Before setting the EType of *packagedElement* in the model it is necessary to create *Package* as a new EClass typed to *NamedElement*.

Further examination of the XMI structure reveals there can be packages with no names within a **<packagedElement…**> so we also create a *packagedElement* as an EReference within the EClass Package with EType *Element*. Two other parameters that need to be set in the Ecore model are *Containment* and *Upper Bound*. Containment refers to whether the element is contained within the enclosing structure or if it is a reference to some other element. The XMI element **<packagedElement…**> is contained within its enclosing structure so we double-click on the *packagedElement* to call up its property sheet and set *Containment* to true. The upper bound refers to how many similar elements there can be within the tagged structure. Since there can be an unbounded number of < **packagedElement…**> we set the *Upper Bound* to -1 which indicates that there can be 0 to many similar structures.

Continuing to examine the XMI file we find that the SysML structures like blocks begin with the same tag <**packagedElement…>** but have a type **uml:Class** and can be contained within a **<packagedElement…**>. In similar manner, ports for the block are tagged as **ownedPort** XMI elements with type **uml:Port** and can be contained within a SysML block. So we add *ownedPort* as a child EReference within the EClass *Class* and EType it to *Port*. The model element for *Port* is elaborated in the Ecore model based on the XMI structure with the relevant attributes and structures found within the beginning and end of the tag **ownedPort**.

The AADL profile extends SysML with additional AADL constructs by adding stereotypes to base UML constructs. Their definitions are tagged in the XMI file as <**AADL\_Elements…>** as shown in Figure 2. Their definition also includes their type (e.g., data) and the attribute **base\_Class** pointing to their base UML type. Since these are not within the beginning and end of the <**packagedElement…**>, they must be defined in a new *Epackage* that has name *AADL\_Elements*. For example, the <**AADL\_Elements:data …>** XMI element is defined in the Ecore model as the *data* model element in the *AADL\_Elements* package with a child *base\_Class* EReference with an EType of *Class*.

The SysML profile is handled in a similar manner, with SysML constructs defined in the *SysML* Epackage.

## Generating the XMI Parser

An XMI parser is created from the Ecore model. To generate the parser it is first necessary to create a generator (.genmodel) model from the Ecore model. A tutorial on how to create the generator model can be found at <http://eclipsesource.com/blogs/tutorials/emf-tutorial/>. To create the generator model the following steps should be followed:

* Right click on the model folder in the Eclipse Package Explorer and select New → Other. In the “Select a wizard” dialog, expand Eclipse Modeling Framework and select EMF Generator Model, then click Next.
* Enter the name for the generator model (it must end with .genmodel) and select Next.
* On the next screen select the Ecore model importer and select Next. On the following screen, select Browse Workspace and navigate to and select the Ecore model. Select Next, then select Finish. This should create the generator model.

Once the generator model is created, double click on it to open it in an editing window, then right click on the model root and select Generate Model Code as shown in Figure 4. This will generate the source code for the parser in the src-gen folder for the project.

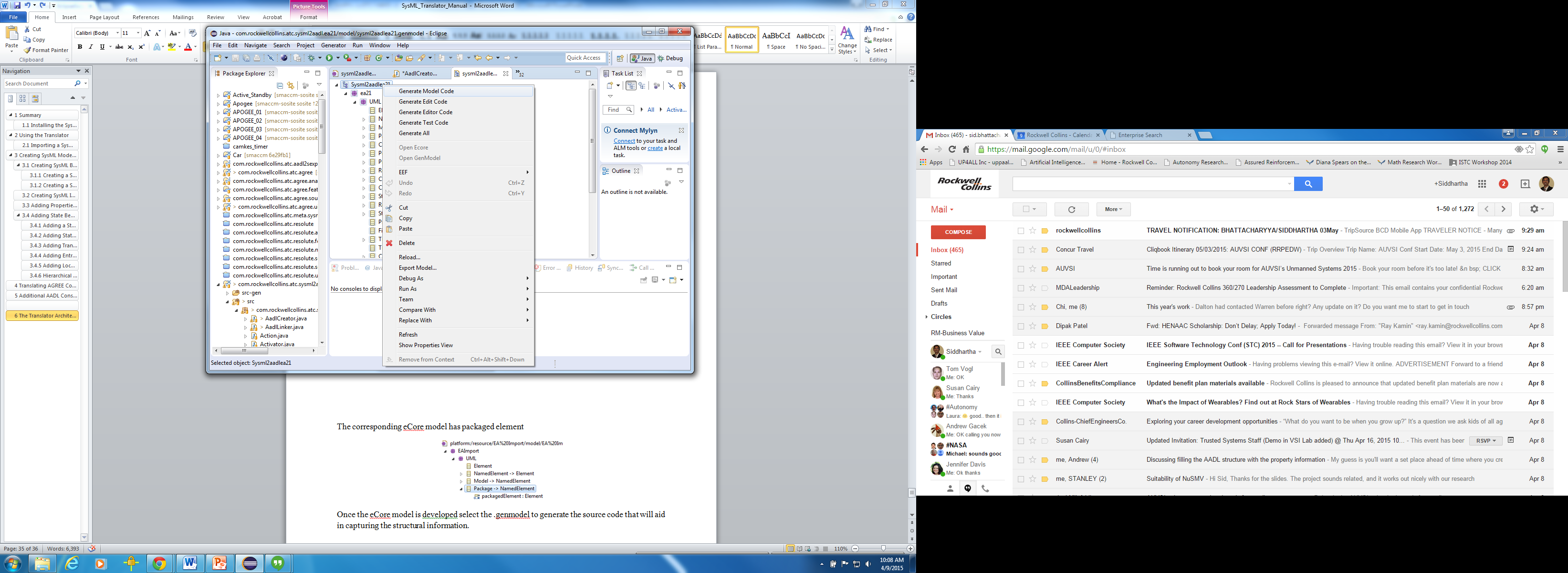


Figure 4 - Generating the XMI Parser

Figure 5 shows a portion of the Ecore model on the left and a fragment of the parser on the right. As would be expected, the parser code closely matches the Ecore model. The NamedElement Java interface class contains methods to get and set the child elements defined in the Ecore model. For example, the *ownedPort* EReference generates a *getOwnedPort()* method that returns a list of ports associated with a component since a subcomponent can have 0 or many owned ports (recall that this was specified in the Ecore model by setting the Upper Bound of the EReference to -1. Similarly there can be list of state machines, list of connections and other list based on the possibility of having 0 or many. In contrast, since the current version of the translator only allows there to be one realization relationship between an implementation and its type, the Upper Bound was set to 0, generating get and set methods that accept and return a single realization value.

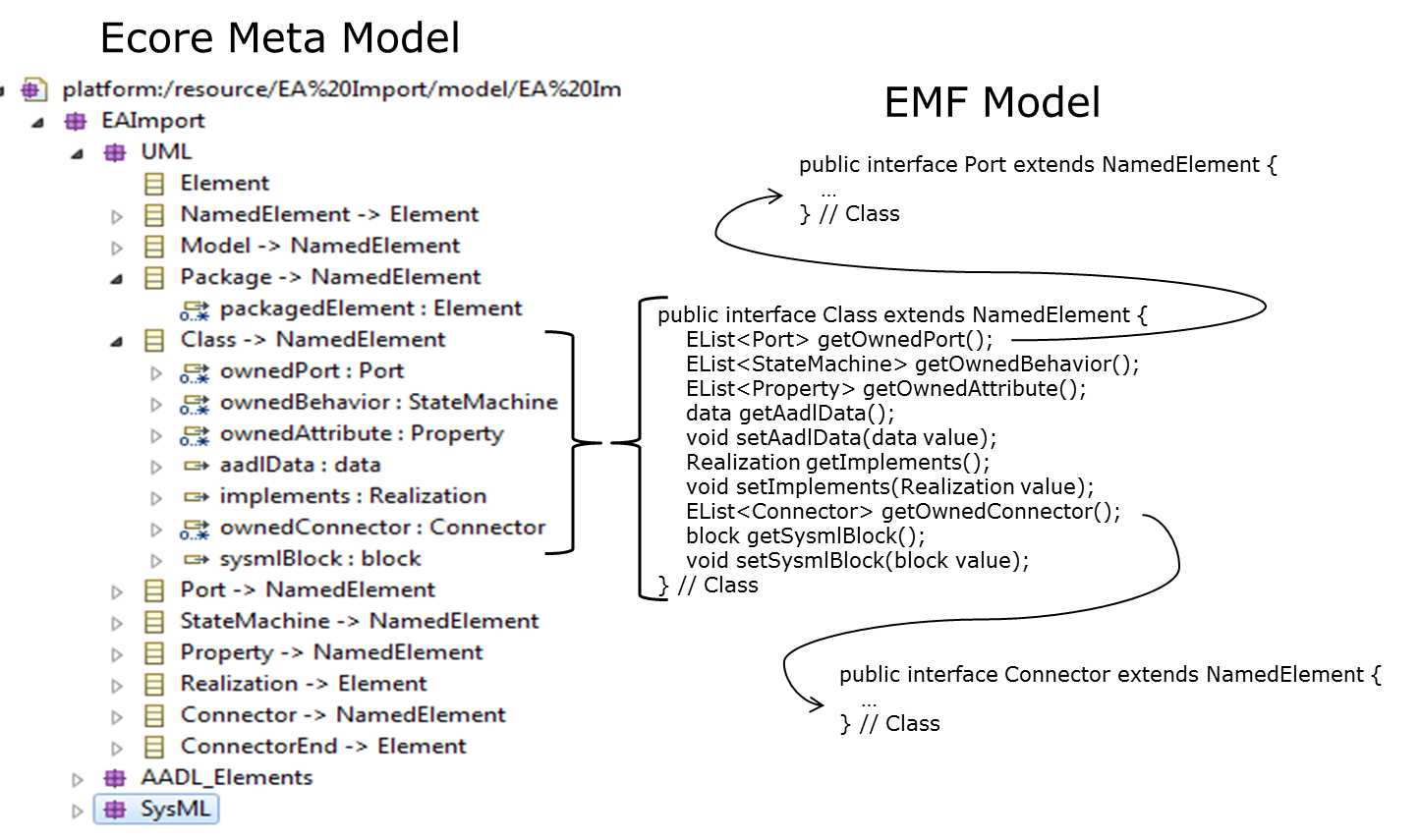


Figure 5 - Ecore Model and the XMI Parser

Once the ecore model is developed and the XMI parser is generated you can check if there are any errors in the model and if the model is extracting the correct information. This can be done by creating a new eclipse configuration. Within the Main tab of the new configuration, provide the Project name as *com.rockwellcollins.atc.sysml2aadl.ea21* and the Main class name as *com.rockwellcollins.atc.sysml2aadl.Main*. Now within the arguments tab provide the location of the xml file as shown. For example: "C:\Projects\Examples\Pilot\_Flying\EA\Pilot\_Flying.xml". This will generate a file in the same location with output attached to its name. For the given location and filename above it will generate the file *Pilot\_Flying\_Output.xml* Now you can open this file to see if you are getting the relevant information as modeled in the ecore. If there are errors in the model it will be displayed in the console.

## Generating AADL

The XMI parser can now be merged with Java source code that implements the SysML to AADL translator. The bulk of the translator is implemented in the Creator and Linker classes as indicated in Figure 6. The Creator traverses the EMF model to create AADL java objects without establishing the links between them. The Linker then uses a map between the AADL and the EMF objects to fill in the links between the AADL objects.

The Creator constructs the AADL objects by making calls to the API provided by OSATE. As it does this, it updates a map that points from each AADL object to the corresponding EMF structure. The creator also maintains a map about the AGREE contracts, a map about any state machines, a map about EMF structure for SysML types (Block Definition Diagram) and implementations (Internal Block Diagrams) without realization relation and a map for variables used in state machines.

The objects presently created are AADL ComponentTypes, Component Implementations, Ports, Subcomponents and Data. Any previously existing AGREE contracts are read from a file in the contracts folder (colocated with the SysML model). State machines are translated into AGREE contracts and inserted into the appropriate AADL component.

The Linker scans through the EMF/AADL map to identify each AADL object and then links it to the appropriate AADL objects by mapping the links from the EMF object to the corresponding AADL objects for those links. The linker first identifies and generates all the required packages, then associates Ports with Component Types, Component Types with Component Implementations, Subcomponents within a Component Implementation, and the Connections among the Subcomponents.

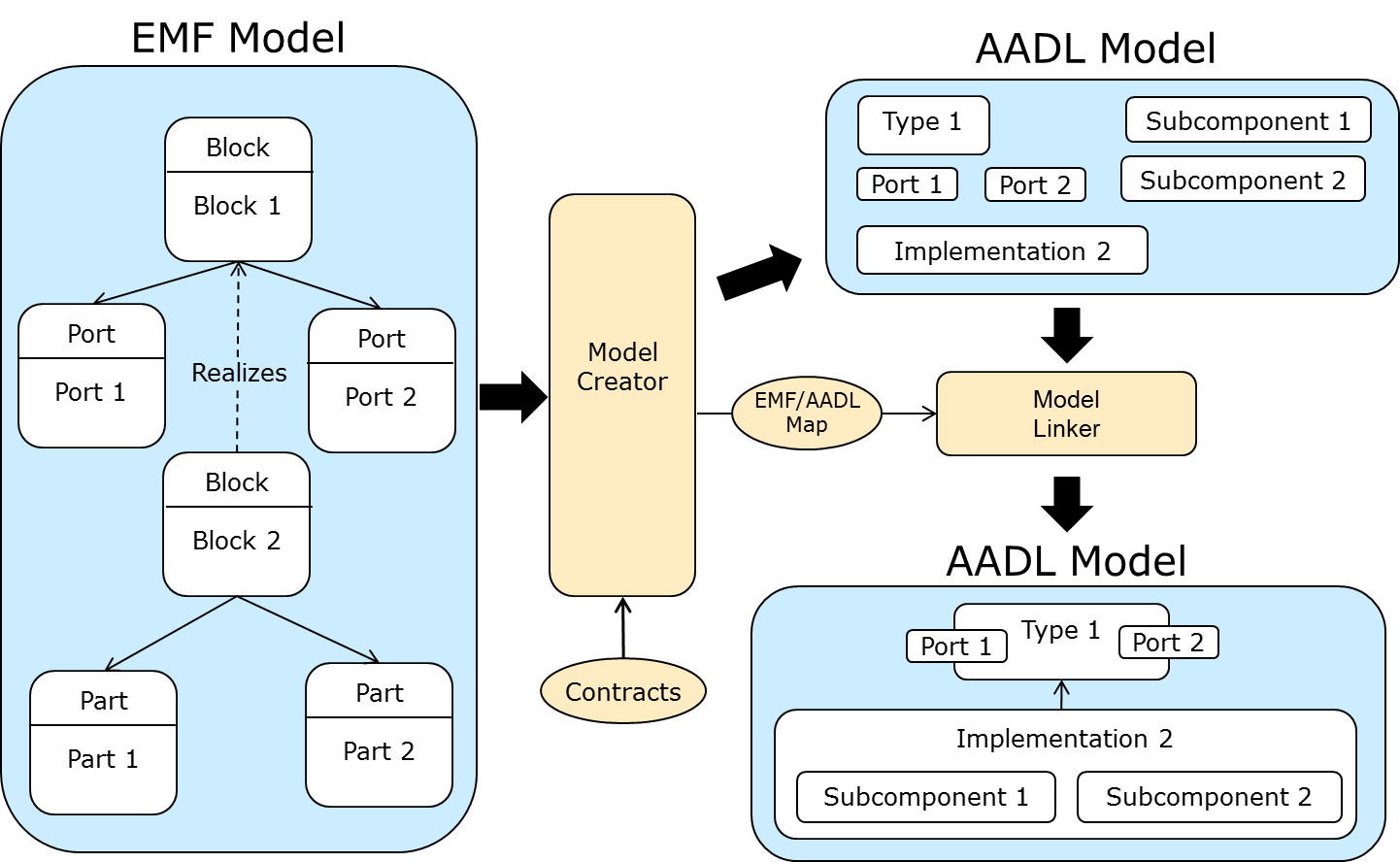


Figure 6 - Translating EMF model to AADL Objects

## Translation of State Machines

Since state machines describe the behavior of a SysML object, state machines are translated into AGREE contracts. This translation is described using the example state machine shown in Figure 7. 

Figure 7 – Example State Machine

States are translated into Boolean variables indicating if the state is active (Figure 8). Meta-states, such as *This\_Side\_Running*, are defined to be the disjunction of their child states.

**eq** Start **:** **bool;**

**eq** This\_Side\_Failed **:** **bool;**

**eq** This\_Side\_Running **:** **bool** **=** Other\_Side\_Failed **or** Both\_Sides\_Running**;**

**eq** Stop **:** **bool;**

**eq** Other\_Side\_Failed **:** **bool;**

**eq** Both\_Sides\_Running **:** **bool** **=** Init **or** Inhibit **or** Other\_Side\_Active **or**  This\_Side\_Active**;**

**eq** Init **:** **bool;**

**eq** Inhibit **:** **bool;**

**eq** Other\_Side\_Active **:** **bool;**

**eq** This\_Side\_Active **:** **bool;**

Figure 8 - States in AGREE

Since the state machines can only be in one state at a time, guarantees or assertions are used to ensure that only one child state can be active at a time (Figure 9). If the enclosing component is an AADL component type, guarantees are generated. If the enclosing component is an AADL component implementation, assertions are generated.

**guarantee** "Start" **:**   
 Start **=>** **not** **(**This\_Side\_Failed **or** This\_Side\_Running **or** Stop**);**

**guarantee** "This Side Failed" **:**   
 This\_Side\_Failed **=>** **not** **(**Start **or** This\_Side\_Running **or** Stop **);**

**guarantee** "This Side Running"**:**   
 This\_Side\_Running **=>** **not** **(**Start **or** This\_Side\_Failed **or** Stop **);**

**guarantee** "Stop" **:**   
 Stop **=>** **not** **(**Start **or** This\_Side\_Failed **or** This\_Side\_Running**);**

**guarantee** "Other Side Failed" **:**   
 Other\_Side\_Failed **=>** **not** **(**Both\_Sides\_Running**);**

**guarantee** "Both Sides Running" **:**   
 Both\_Sides\_Running **=>** **not** **(**Other\_Side\_Failed**);**

**guarantee** "Init" **:**   
 Init **=>** **not** **(**Inhibit **or** Other\_Side\_Active **or** This\_Side\_Active**);**

**guarantee** "Inhibit" **:**   
 Inhibit **=>** **not** **(**Init **or** Other\_Side\_Active **or** This\_Side\_Active**);**

**guarantee** "Other Side Active" **:**   
 Other\_Side\_Active **=>** **not** **(**Init **or** Inhibit **or** This\_Side\_Active**);**

**guarantee** "This Side Active" **:**   
 This\_Side\_Active **=>** **not** **(**Init **or** Inhibit **or** Other\_Side\_Active**);**

Figure 9 - Guaranteeing Only One Child State Active

Local variables referenced by the state machine are also declared as AGREE variables   
(Figure 10).

**eq** InhibitCount **:** **int;**

**eq** InitCount **:** **int;**

Figure 10 – Local Variables

The initial state is set by a guarantee ensuring that in the first step the *Start* state is active (Figure 11). Note that this guarantee says nothing about the active state on subsequent steps.

**guarantee** "Initial State" **:** Start **->** **true;**

Figure 11 – Specification of the Initial State

Transitions are translated into Boolean predicates and a set of assignments that must hold when the transition is true. For example, transition T4 from *This\_Side\_Running* to *This\_Side\_Failed* is defined as shown in Figure 11.

**eq** T4: **bool = (false -> (pre(**This\_Side\_Running) **and** ThisSideFailed**));**

**guarantee** "T4\_next\_state" **:** T4 **=>** This\_Side\_Failed**;**

**guarantee** "T4\_ThisSideActive" **:** T4 **=>** **(**ThisSideActive **=** **false);**

Figure 12 – Specification of Transition T4

The Boolean variable T4 is specified to be false on the first step (all transitions are false on the first step) and on every subsequent step it is true if the state variable *This\_Side\_Running* was true in the previous step and the port input *ThisSideFailed* is true on this step. That is, the transition is true only if the source state was active and the transition guard is true.

The guarantees (or assertions if the enclosing component is an AADL component implementation) then specify the new active state and the values of any local variables or output ports changed by the transition. In Figure 12, the state *This\_Side\_Failed* becomes active and the output port *ThisSideActive* becomes false. The previous state (*This\_Side\_Running* and whichever of its child states was active) is cleared by the definitions of the states and the guarantees in Figure 8 and Figure 9.

Transitions into meta-states such as T3 and TTSR2 are flattened into a single composite transition. For example, the transition from *This\_Side\_Failed* to *Init* is defined as shown in Figure 13. Note that the last guarantee ensures that the entry condition of the *Init* state is enforced.

**eq** T3\_TTSR2\_TBSR1**:** **bool** **=** **(false** **->**   
 **(pre(**This\_Side\_Failed**)** **and** **not** ThisSideFailed **and** **not** OtherSideFailed**));**

**guarantee** "T3\_TTSR2\_TBSR1\_next\_state" **:** T3\_TTSR2\_TBSR1 **=>** Init**;**

**guarantee** "T3\_TTSR2\_TBSR1\_ThisSideActive" **:** T3\_TTSR2\_TBSR1 **=>**

**(**ThisSideActive **=** **Get\_Property(this,** QS\_Properties**::**Primary\_Side**));**

**guarantee** "T3\_TTSR2\_TBSR1\_InitCount" **:** T3\_TTSR2\_TBSR1 **=>** **(**InitCount **=** 0**);**

Figure 13 – Composite Transition into Meta-States

During-transitions are performed while a state is active. For example, the during-transition for the *Init* state increments the *InitCount* local variable. It is specified as shown in Figure 14. Note the additional guards in the transition ensuring that none of the exiting transitions T4, TTSR4, TBSR3, or TBSR2 can be true.

**eq** Do\_Init**:** **bool** **=**

**(false** **->** **(pre(**Init**)** **and** **not** **(**T4 **or** TTSR4**)** **and** **not** **(**TBSR3 **or** TBSR2**)));**

**guarantee** "Do\_Init\_next\_state" **:** Do\_Init **=>** Init**;**

**guarantee** "Do\_Init\_InitCount" **:** Do\_Init **=>** **(**InitCount **=** InitCount**+**1**);**

Figure 14 – During Transitions

Finally, since AGREE is a constraint-based language, it is necessary to add guarantees or assertions that no local variables, output ports, or state changes unless a transition changes them. This is done by writing guarantees that if no transition is taken that can change the object; its value remains unchanged (Figure 15).

**guarantee** "no\_change\_InhibitCount" **:** **true** **->**   
 **(not** **(**TBSR3 **or** TBSR4 **or** Do\_Inhibit**)** **=>**   
 **(**InhibitCount **=** **pre(**InhibitCount**)));**

**guarantee** "no\_change\_InitCount" **:** **true** **->**   
 **(not** **(**T2\_TTSR2\_TBSR1 **or** Do\_Init **or** T3\_TTSR2\_TBSR1 **or** TTSR3\_TBSR1**)** **=>  
 (**InitCount **=** **pre(**InitCount**)));**

**guarantee** "no\_change\_state" **:** **true** **->**

**(not** **(**T3\_TTSR2\_TBSR1 **or** T3\_TTSR1 **or** T4 **or** TTSR3\_TBSR1 **or** TTSR4 **or**   
 TBSR5 **or** Do\_Inhibit **or** TBSR3 **or** TBSR2 **or** Do\_Init **or** TBS7 **or**   
 TBSR6 **or** TBSR4 **or** T2\_TTSR2\_TBSR1 **or** T2\_TTSR1 **or** T1**)** **=>**   
 **(**Start **=** **pre(**Start**))** **and** **(**Stop **=** **pre(**Stop**))** **and** **(**This\_Side\_Failed **=** **pre(**This\_Side\_Failed**))** **and** **(**This\_Side\_Running **=** **pre(**This\_Side\_Running**))** **and** **(**Other\_Side\_Failed **=** **pre(**Other\_Side\_Failed**))** **and** **(**Both\_Sides\_Running **=** **pre(**Both\_Sides\_Running**))** **and**   
 **(**Init **=** **pre(**Init**))** **and**   
 **(**Inhibit **=** **pre(**Inhibit**))** **and** **(**Other\_Side\_Active **=** **pre(**Other\_Side\_Active**))** **and** **(**This\_Side\_Active **=** **pre(**This\_Side\_Active**)));**

**guarantee** "no\_change\_ThisSideActive" **:** **true** **->**   
 **(not** **(**T4 **or** TTSR4 **or** T2\_TTSR1 **or** T3\_TTSR2\_TBSR1 **or** T1 **or**   
 T2\_TTSR2\_TBSR1 **or** T3\_TTSR1 **or** TBS7 **or** TBSR2 **or** TBSR3 **or** TTSR3\_TBSR1 **or** TBSR4 **or** TBSR6**)** **=>**   
 **(**ThisSideActive **=** **pre(**ThisSideActive**)));**

Figure 15 - No Change Guarantees

1. The XMI structural elements are written in bold and the Ecore elements are written in italics. [↑](#footnote-ref-2)