Modern, complex software systems for a specific application domain often display common system design architectures with similar subsystem functionality and interactions, making them suitable for representation by reusable specification architecture. For example, every distributed networked system requires connection management, event detection, de-multiplexing, event dispatching, and composition of dynamically configured services. The similarities between these subsystems can be exploited to create a model-driven system development environment in which generic reusable specifications and models can be tailored for the specific platform, and automatically transformed into software. Modifications to software during operations can be similarly made in the same controlled way that is, starting from a model, validating the change, and finally implementing the change. This paper proposes a domain-specific, component-based framework called as Automated Software in Networked Environment (ASWIN) employs three kinds of models, which are Computation Independent Model (CIM), Platform Independent Model (PIM) and Platform Specific Model (PSM). Ontologies are adopted to represent CIM. The PIM includes a metamodel called NetProc(Networked Process) for specifying system-level architecture, and a visual graphical Domain-specific modeling language called NPML (Network Programming Language). With the instrumentation of NeProc as a Generic Modeling Environment (GME) metamodel, a distributed networked system becomes composition of NetProcs, Sub-Systems, Shared Objects and their interconnections. As part of the PSM, the exported model in XML can be parsed and a proper intermediate structure can be constructed. Applying appropriate synthesis techniques and by reusing components, the software generation is automated. Operating system portability is maintained by building the framework atop the Adaptive Communication Environment (ACE), which has been ported to most platforms including Linux, Windows, and Macintosh. The proposed framework supports customization of a distributed networked application systems and is found to increase the software productivity.

Keywords: Frameworks, Model Driven Architecture, Model Synthesis, Networked Systems, Ontology.

1. INTRODUCTION

In comparison, today’s networked environment is very complex. It includes equipment and software from multiple vendors, multiple protocols, distributed file systems and system services, and a “backplane” with both hardware and microcode components which is constantly being modified by people with widely varying skills and responsibilities. This complexity has resulted in decreased support effectiveness which in turn leads to: decreased quality of service; inability to support existing and new applications; and increased downtime, users’ time wasted and security exposures. Example of such distributed networked applications systems are WWW, Email, a distributed database, or a spreadsheet accessing a distributed file system.

Due to the huge complexity of these systems, it is required to specify precisely what a software component should do and how it should behave. Model-driven development (MDD)[1] is a promising paradigm for tackling system composition and integration challenges. MDD elevates the abstraction level of software development and bridges the gap between technology domains by allowing domain experts (who may not be experts in software development) to design and build systems. MDD deals with the complexity of modern systems by constructing models: abstract representations of structure or behavior[2]. We use these models to understand, to describe, and to examine the properties of the systems they represent; we may even use them to generate components of the systems themselves. Using this technique, complex systems are modeled at different levels of specificity. As the development program proceeds, the model undergoes a series of transformations, with each transformation adding levels of specificity and detail.

A primary example of MDD is the Model-Driven Architecture (MDA) initiative of the Object Management Group (OMG) [3]. MDA proposes a software development process in which the key notions are models and model transformation, where the input models are platform independent and the output models are platform specific and can be transformed into a format that is executable. The several abstraction levels are integrated by model transformation techniques. MDA introduces an approach to system specification that separates the views on three different layers of abstraction:
high level specification of what the system is expected to do (Computation Independent Model or CIM), the specification of system functionality (Platform Independent Model or PIM) and the specification of the implementation of that functionality on a specific technology platform (Platform Specific Model or PSM)[4]. The proposed framework has been designed and implemented based on MDA approach.

Specifically, following contributions are made:

1. The first abstraction level defines a Computational Independent Model - CIM, called NetON that is free of any reference to the computational and communicational concepts (system’s data structures or low level algorithms). Such models can be seen as abstractions of the requirements models for distributed networked systems.

2. The second abstraction level is the Platform Independent Model - PIM that is free of any specific platform detail (operating system, specific frameworks, persistence technology, etc.). It is focused on the relation between the elements, what data structure can be used to represent their relationship and what are the constraints that should affect such relationships. A meta-model has been modeled as NetProc and has been instrumented in GME[5].

3. The framework ASWIN also provides a domain-specific modeling language (DSML) called NPML that enables the developer to graphically create the model of distributed networked system for which the software is to be generated.

4. Code generator to translate the PIM to code/API in Adaptive Communication Environment (ACE)[6] as the third abstraction level, Platform Specific Model – PSM.

5. Operating system portability is maintained by building our framework atop the ACE, which has been ported to most platforms including Linux, Windows, and Macintosh.

The paper is organized as follows. Section 2 gives an overview of the framework and related terms. Section 3 describes the MDA approach applied to the development of the framework ASWIN along with involved tools, environments and techniques. Section 4 illustrates a case study of generation of a chat server code automatically as an example. Some final remarks and an outlook on future work which concludes the paper.

2. MODEL DRIVEN APPROACH (MDA) BASED FRAMEWORK

The OMG’s MDA specification[17] establishes the existence of three main abstraction levels (or visions), that can be transformed from one to another linearly. The meta-modeling approach of framework components in the component design aspect, and the code generation techniques for component implementation aspect are shown in Fig. 1.

The meta-models of the framework components are represented as the UML[7] class diagrams provided by the meta-modeling tool GME, a configurable toolkit that supports the creation of domain-specific modeling and program synthesis environments. For implementing the system, the required basic components are fabricated i.e. coded in its application domain, using a network API (e.g. Socket API components in ACE framework). These components are wrapped into the framework components and stored under the repository for reuse. Components with more functionality can be generated using these wrapped components. They also can be stored under the repository as templates for reuse or for developing similar components. Consequently, these components are used to generate the code, which could be used for simulation or deployment in the real environment.

2.1 Ontology

The development of software systems is a complex activity which may imply the participation of people and machines (distributed or not). Therefore, different stakeholders, heterogeneity and new software features make software development a heavily knowledge-based process. To reduce this complexity, the application of ontologies[8] might prove useful. Ontologies allow for the definition of a common vocabulary and framework among users (either human or machines).

Although there are yet many definitions for ontology, the most general and complete one is “an explicit and formal specification of a shared conceptualization”. In brief, ontology aims in defining a set of concepts, properties, and their axioms that provide rules, which govern them.
According to Gruber [9], an ontology defines a set of representational primitives with which to model a domain of knowledge or discourse. The representational primitives are typically classes (or sets), attributes (or properties), and relationships (or relations among class members). An additional value of the ontology-driven approach is that it enables people without an extensive software engineering background to effectively participate in the development or modification of software[10].

2.2 Model, Metamodel and Metamodel
Models are the key artifact of MDD. “A model is a simplification of a system built with an intended goal in mind. The model should be able to answer questions in place of the actual system.” While this serves as a starting point, Kleppe et al. [11] gives a definition even more directed to MDD. “A model is a description of a (part of) systems written in a well-defined language. A well-defined language is a language with well-defined form (syntax), and meaning (semantics), which is suitable for automated interpretation by a computer.”

Metamodels define how a model can look alike. The word “meta” is Greek and means “above”, therefore the term metamodel can be interpreted as a model describing another model[12][13]. This can be more precisely formulated as: a metamodel defines the constructs and rules usable to create a class of models. Notice that a metamodel is itself a model. If this is true the metamodel has also to be defined in a “language” (needs a metamodel). For that reason the metametamodel is introduced, allowing to specify metamodels. The question which can be raised is how the metametamodel is defined. To avoid an infinite stacking of meta levels, metametamodels are often specified self reflexive and therefore the metamodel of the metametamodel is the metametamodel itself. Most approaches implementing MDD define a three level meta stack - model, metamodel, metametamodel. The ASWIN framework proposes and implements a metamodel called NetProc which can be used to define models or elements involved in a distributed networked systems.

2.3 Generic Modeling Environment (GME)
GME is an open source, visual, configurable design environment for creating DSMLs[14] and program synthesis environments. One of GME’s unique features is that it is metaprogrammable—that is, GME is used not only to build DSMLs but also to build models that conform to a DSML. In fact, MetaGME, the GME environment used to build DSMLs, is itself built using another DSML – or metamodel. GME’s metamodel offers stereotyped concepts such as Atom (elementary object), Model (which can have inner parts and structures), Connection(relationship between two objects within one model), Reference, Attribute, Set (similar to a UML aggregation) and other FCO (first-class objects). Predefined data types such as field, enum, and bool are also available. An Aspect can be used to control the visibility of elements in the editor. OCL constraints can be added to increase the precision of the paradigm and to enable syntactical validation of user models in the target DSML editor.

2.4 Domain Specific Modeling Language (DSML)
A domain-specific language provides domain experts with a familiar abstraction for creating computer programs. A domain-specific language allows a modeler to describe the system in terms of the domain rather than in terms of traditional computer languages. Domain-specific modeling environments (DSME’s) are the interface for domain experts to program using a domain-specific modeling language (DSML). The DSME[15][18] provides domain-specific constructs that are associated with one another to describe a computer-based system. As more and more domains embrace computers, programmers are tapping into this power by creating their own languages fitting the particular needs of the domain.

Graphical domain-specific modeling languages are even more appealing for non-programmers, since the modeling language constructs are automatically transformed into applications through a special compiler called a translator. The value of domain specific modeling languages is about getting productivity and flexibility from abstraction. Instead of trying to pick a single language with a large enough domain, pick a set of languages whose domains form a covering set of the domains of relevance to the problem at hand, ensuring that the languages. The cognitive distance from the problem domain to the software domain is drastically reduced, leading to fewer defects, less effort and more productivity, etc.

3. THE ASWIN FRAMEWORK
The developed framework is component-based, with its domain as distributed networked application systems called as ASWIN. It supports computation and communication within the system and communication among the networked systems. It also provides modeling techniques in the solution space of distributed networked application systems.

3.1 Computation Independent Model (CIM)
The purpose of the Computation Independent Model(CIM) is to capture a domain with its concepts and properties. Typically, two viewpoints of domain modeling can be distinguished. Concepts are represented in form of hierarchies – called the information viewpoint in MDA. Behavior is represented in a process-based form – called the process viewpoint in MDA, based on distributed processing concepts.
3.1.1 Conceptual Model

A generic conceptual system model called Netwroked Process (NetProc) has been designed as shown in Fig. 2. The system model contains component structures, interaction and hierarchy. The framework employs a two-level architectural model to specify the system architecture. The first level (system level) contains a set of intercommunicating subsystems, processes and the shared objects. At the second level (process level), a process contains communication points, communication parameters and functional services. Communication part is described with three kinds of details. First of them is the binding data, which could be the IP addresses and port numbers, URLs, and I/O addresses. Second one involves the connection types as direct and indirect. Direct connection is established between the NetProcs directly and can be of active or passive type. Indirect connections are between the NetProcs and shared objects. The connection management, event detection, de-multiplexing, and event handler dispatching is supported by the last part called as behavioral logic.

The modeling in the framework is based on two perspectives – component aspect and procedural aspect. The component structures in the application are described using the generic object-oriented approach. The procedural structures (computation and communication) are portrayed by action sequences. Action sequences can be represented as flowcharts, which consist of action (i.e. action blocks, tasks, method calls and communication calls), action controls and logic flow. The architectural view of the NetProc for its corresponding conceptual view is shown in Fig. 3.

3.1.2 NetOn: Networked Systems Ontology

Ontologies are used as inputs for developing the software architecture, used in the specification of the system and for discovery of appropriate software components. A domain ontology explains the types of things in that domain. The CIM does not show details of the structure of the system. In software engineering, it is well-known as a domain model specified by domain experts. This is very similar to the concept of ontology. That’s why we choose ontology support for our CIM. The Ontology designed for the ASWIN framework has been named as NetOn. The taxonomy and slot details of concepts of NetOn is shown in Table 1.

<table>
<thead>
<tr>
<th>Concept</th>
<th>Taxonomy</th>
<th>Attributes</th>
<th>Role</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td>NameSpace*,</td>
<td>Author,</td>
<td>Application</td>
</tr>
<tr>
<td></td>
<td>System*</td>
<td>Version</td>
<td>Software</td>
</tr>
<tr>
<td>NameSpace</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>System</td>
<td>Process*,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Communication</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Connection</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Process</td>
<td>Info_Component*,</td>
<td>IncludeFiles,</td>
<td>Subsystem software</td>
</tr>
<tr>
<td></td>
<td>Task*</td>
<td>Reference</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Namespace</td>
<td></td>
</tr>
<tr>
<td>Info-Component</td>
<td>Object*,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Communication</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Point</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Task</td>
<td>ActionSeq</td>
<td>Flow type</td>
<td>Separate thread</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(Thread)</td>
<td></td>
</tr>
<tr>
<td>Action_Seq</td>
<td>Action_Block*,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Action_Control*,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Task*, Method_Call*,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Communication_Call*,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Flow_Connector</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Action_Block</td>
<td>Code/Code file</td>
<td></td>
<td>Block of statements</td>
</tr>
<tr>
<td>Action_Control</td>
<td>Logic_Control*,</td>
<td></td>
<td>Conditional statements</td>
</tr>
<tr>
<td></td>
<td>Iteration</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Logic_Control</td>
<td>If</td>
<td>Condition</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>expression</td>
<td></td>
</tr>
<tr>
<td>Iteration</td>
<td>For, Repeat_until,</td>
<td>Initialization,</td>
<td>Loop statements</td>
</tr>
<tr>
<td></td>
<td>Repeat-if</td>
<td>condition,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>increment</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2: NetProc Design (Conceptual)

Figure 3: Architectural Design of NetProc.
3.1.3 CIM to PIM (Ontology to System Model in DSML)

Even though we focused much on the PIM and PSM levels, proposing the required ontology as suggested input form to the PIM level, fulfills the MDA approach[20]. The proposed ontology can be used in the computational independent model of the system in the form of manual scripts. Several editors (ex. protégé)[16] exist to design this ontology using an editor and can be stored in an intermediate form. This form can be translated to ASWIN framework’s DSML, i.e. NPML by an appropriate model translator tool. In future extensions part, we feel that this form can be translated to the PIM level of NPML icons.

3.2 PIM – Platform-Independent Model

The platform-independent model (PIM) shifts the focus from the computation-independent capture of the domain to a focus on architectural constraints imposed by the computational environment. Architectures and processes are the key aspects at this modeling level. The architectural focus is on services, their architectural configuration, and interaction of processes.

Architectural configuration addresses the interaction processes between different components of a software system.

3.2.1 Meta-level Definitions of ASWIN Components

Metamodels support abstraction and generalization that aid in identifying problems that can use the same solutions. Mappings provide bridges between those alternative solutions. Modeling a system makes it much easier to communicate information about the system between developers. However, the creation of the right model abstractions is harder than developing concrete programs. One of the most important concepts in modeling development is the idea of metamodel. It is in the metamodel that most of the model transformation approaches rely on. They manipulate the instantiations of a metamodel by means of the metaclasses. The CIM model NetProc has been instrumented as a meta-model in GME as shown in Fig.8. Few other metamodels are shown in Fig.4, Fig.5, Fig.6, Fig.7.

Figure 4: Metamodel of Action Sequence
Figure 5: Metamodel of Action

Figure 6: Metamodel of Object Dependency

Figure 7: Metamodel of Object Class
3.2.2 The Network Programming Modeling Language (NPML) Domain-specific Modeling Language

The key idea behind a DSML is its ability to capture domain elements as first-class objects.

DSMLs are defined visually in GME. With the mapping of domain elements to GME concepts, UML class diagrams represent the DSML elements. With GME, users can customize the visualization of DSML elements — that is, its concrete syntax — using a decorator, a component written in a traditional programming language that implements a set of standard callback interfaces. Once a decorator is registered with GME, the environment invokes the callbacks whenever it needs to display the element. These are shown in table 2.
After defining the DSML’s elements and its associated static semantics, the developer instructs GME to generate a customized DSML environment using a process called meta-interpretation. This process takes the definition of the DSML from the previous step; runs a set of standard transformations ensuring consistency (as does a traditional compiler); and creates a paradigm file defining the DSML, which is then registered with GME. It is now possible to create models conforming to the DSML using GME. Thus NPML, the Platform-Independent visual language could be used for model-based system development in the domain of distributed networked systems.

### 3.3 PIM – Platform-Independent Model

It usually consists of a platform model that captures the technical concepts and services that make up the platform and an implementation-specific model geared towards the concrete implementation technique. The framework’s code-generation mechanism includes parsing and synthesis.

#### 3.3.1 Application System Analysis: Parsing

After exporting the application system model in XML format, it will be parsed to generate a parse tree called as Object Tree with models as non-leaf nodes, and atoms as leaf nodes. The exported XML file embeds the model and meta-model-specific information. The ACE-XML, a validating XML parser built in C++, has been used for parsing.

The main steps in parsing are:

**Step 1. XML element identification:** An object tree has to constructed with each node representing a GME element. The attributes associated with the node (Model Object) are Object ID, Parent ID, Associated Model, Name, Kind, Reference Id, Children, Position, Connection properties, and Model Attributes which are provided to the element as input by the user.

**Step 2. Generation of Object Tree:** When the XML parser identifies an XML element, a model object to the tree corresponding to the identified element is created. For every inner XML element of the current element, a new model object is created and added into the children of the current model object. This model object is then identified as the current object. The object tree is created using a depth first traversal. The generated object tree is used to generate code from code generating tree.

**Step 3. Storing the Object Tree in Object Pool:** The object pool consist of all the model objects in one set and the base objects in another. A method is defined in the object pool, to access any model object in the whole model using its ID. Methods are also defined to list the model objects to which, a given model object is connected, for a specified type of connection. Base objects form the root nodes of the whole model.

#### 3.3.2 The Pseudo-code for the Object Tree Generation(Parsing) is as follows

```cpp
GenerateObjectTree(XMLmodel) {
    for each element in XMLmodel, do {
        E = element;
        modelObject = createModelObject();
        modelObject.SetAttributes(E.Object ID, E.Parent ID, E.Name, E. Kind, E. position);
        if E. Kind is Connection modelObject. Connection Properties = E. Connection Properties;
        for each innerXMLmodel in E, do
            modeObject. Children = GenerateObjectTree(innerXMLmodel);
            BaseObjects.add(modelObject);
        ObjectPool.add(modelObject);
    }
    return BaseObjects;
}
```

Hence, in any distributed networked application system model, the system becomes the root of the parse tree and the different components of the system and their interconnections form the children of the root. The component could even be a sub-system, which in turn
has its corresponding component nodes and interconnections as children. In the class model, the namespace forms the root, with the sub-namespaces and classes as children. The children of classes include the methods, communication methods, and properties. The information of each leaf node (NetProcs) of the generated tree will be stored in the respective files as the intermediate representation. These files representing the NetProcs will be stored in the folder corresponding to the containing System. This intermediate representation of the system is in a platform-independent form, which allows its implementation in any targeted platform. This file-based representation of the system allows the reusability of the components at a later stage.

3.3.3 Synthesis of the Application System: Automatic Code Generation

The generated intermediate file which contains the Object Tree is used to generate an equivalent tree, with the nodes capable of generating the code for the object that it represents. For example, for a model object that represents a class, the new tree would contain an associated node, which generates the code for the class[19]. As each node in the Object Tree has its corresponding node in the code generating tree, they are linked to each other. The complete model is represented by the Code Generating Tree.

The algorithm for the code generator is summarized by the following pseudo code.

```
GenerateCode(model) {
    for each system in model, do {
        for each sub-system in system, do //child node of the system
            GenerateCode(sub-system.model);
        for each shared-object in system, do //child node of the system
            if shared-object.ClassFOUND() is false {
                classFile = GenerateClass(shared-object.Class);
                namespaceFolder = CreateFolder(shared-object.Class.Namespace);
                namespaceFolder.insert(classFile);
                Repository.Add(namespaceFolder); }
            for each process in system, do //child node of the system
                for each component in process, do {
                    if Repository.Find(component) is false {
                        classFile = GenerateClass(component.Class);
                        namespaceFolder = CreateFolder(component.Class.Namespace);
                        namespaceFolder.insert(classFile);
                        Repository.Add(namespaceFolder); }
                    InstantiateObject(component); IncludeReference(component); }
                for each dependentSharedObject in process, do {
                    InstantiateObject(dependentSharedObject);
                    IncludeReference(dependentSharedObject); }
            processActionCode = GenerateActionCode(process.ActionSequence);
            processFile = GenerateProcess(process);
            systemFolder = CreateFolder(system);
            systemFolder.Add(processFile); }
        Repository.Add(systemFolder); //Create a folder in repository for the namespace of the created class }
}
```

The procedure GenerateClass() is used to generate the code for each class. Each class is separately stored in a file, under the folder corresponding to its namespace, stored in the repository. The procedure GenerateActionCode() generates the code from the Action Sequence, and has a key feature that the code generated by it can be for multi-threaded application, as per the design specifications. This is made possible by linking the generated code to a module called 'ActionSequence Manager'. Operations for registering the Actions, Joins and the order they follow are defined in ActionSequence Manager. The execution of the Action Sequence is then left to the ActionSequenceManager, which invokes the threads corresponding to the actions and joins in the specified order. Hence, the scheduling of actions is appropriately done in every Action Sequence. The procedure GenerateActionCode() merges sequential nodes to a single node. This optimization improves the performance by removing unnecessary threads. The generated code could be stored in the repository along with the model for reuse. This code can be provided to simulation tools or to build distributed networked application system software.

While the code generator supports synthesis of application-behavior code, it is feasible to link in externally supplied application-behavior code, which can be either typed online or with inclusion of a code file.

4. CASE STUDY : APPLYING NPML TO CHATSERVER EXAMPLE

We demonstrate the use of ASWIN’s NPML by designing a chat server system model. The system consists of two subsystems, client and server as shown in Fig.9 (a). The computational and communicational logic of the client system has been programmed in NPML with two tasks of Send-Data and Receive-Data as shown in 9 (b). NPML of the two tasks of client system is shown in Figures 9 (c) and 9 (d). The Server system is coded in NMPL with task of Interaction-with-client in Fig. 9 (e). The task Interaction-with-client and its two tasks Receive-
Message and Send-Message in NPML are shown in Figures 9 (f), 9 (g), and 9 (h).

5. CONCLUSIONS
A model-driven framework aiming at the software development in the domain of distributed networked application systems has been developed. The framework provides the modeling methods for domain-specific features of a networked system in different aspects, including component structures and interaction, system functionality, etc. A complete formal definition of the framework components carried out in GME consists of a metamodel specified as NetProc. It is now possible to create models conforming to the DSML using GME. Graphical domain-specific modeling languages are more appealing for non-programmers, since the modeling language constructs are automatically transformed into applications through a special compiler called a translator or code generator. NPML as DSML, productivity and flexibility from abstraction has been achieved. Automatic synthesis of the application system has been done by parsing of models and generating the source files with code generation algorithm. It is an extensible architecture for integrating code generation capabilities, and other verification tools. The final result of research is the development of networked application system software from reusable and re-configurable components. And the
approach has been proved to produce the required results by implementing few sample networked applications like internet super server, file transfer servers, email servers and routers. Networked applications can be created consistently between specification and implementation and with less effort than manual coding. The designed framework provides effective support for the software architect through service architecture modeling on an abstract level. The software architects benefits from semantic modeling and reasoning, improved maintainability, and automated generation of potentially a range of different platform-specific implementations.

Even though we have implemented the PIM and PSM levels of ASWIN, proposing the required ontology as suggested input form to the PIM level, fulfills the MDA approach. At present, the proposed ontology NetON can be used in the computational independent model of the system in the form of manual scripts. By the usage of existing editors to design this ontology, the system model can be converted into intermediate form. This form can be translated to ASWIN framework’s DSML, i.e. NPML by an appropriate model translator tool. These extensions enable ontology-based creation of model elements, discovery and reuse of both the software components and partial models through a repository mechanism.

REFERENCES