Domain Library Preprocessing in MWorks - a platform for Modeling and Simulation of Multi-domain Physical Systems Based on Modelica

Wu Yizhong, Zhou Fanli, Chen Liping, Ding Jianwan, Zhao Jianjun
CAD Center, Huazhong University of Science and Technology
Wuhan, Hubei, China: 430074
Cad.wyz@gmail.com, Chenlp@hustcad.com

Abstract

Modeling and simulating with Modelica-based platforms for multi-domain physical systems need support of diversified domain libraries. Modelica Standard Library (MSL) is a free, reusable library released by Modelica Association, which comprises definitions of basic classes belonging to many domains such as mechanical, electrical, hydraulic, control, multi-body, thermodynamics, etc. User can also define or extend his special library. These libraries usually are stored at disk with mo text format (or binary encrypted format). They only record text information of the defined classes, and do not save their structured information. They will be loaded when system starts up, and they can be reused as extends classes or component instances when modeling. When the current main class is being checked or translated, the referenced classes (including extends and component classes) from domain library will be parsed first.

Loading mo format files involves searching of a great many files from the library directory. At the same time, lexical and syntax analysis is performed. So the loading process is time-consuming. Likewise, parsing the referenced classes from domain library when parsing the current main class is also a time-consuming process. So this paper studies a so-called domain library preprocessing technology which can greatly improves loading speed of domain library and parsing speed of the main class. The idea of this approach is that: parsing all the classes within domain library once, building their structured information of document object models, and then saving them to a binary file. When the system starts up, the binary file is automatically serialized to construct structured information of the classes which were defined originally in domain library. This approach is implemented in MWorks – another independent platform for modeling and simulation of multi-domain physical systems based on Modelica language.

Keywords: Modeling and Simulation; Modelica; Domain Library; Preprocessing; Document Object Model

1 Introduction

Modelica language has been developing rapidly during these years[1]. Applications of modeling and simulation based on Modelica have also been emerging in endlessly. However, software implements based on Modelica language can be counted on your fingers. Dymola[2] and MathModelica[3] are currently the most two important and successful commercial software. In addition, other Modelica-based software which are been implemented include OpenModelia[4], MOSILAB[5] and SCICOS[6], etc.

Having seen the daylight of Modelica simulation language in future, we have been bending ourselves to implementing an integrated development environment based on Modelica from 2003. And now our work has come into being an alpha stage product – MWorksV1.0a.

MWorks is another Modelica-based platform for modeling and simulation of multi-domain physical systems, which wholly integrating a modeler, a translator, an optimizer, a solver and a postprocessor. Like other platforms based on Modelica such as Dymola, MathModelica, etc., MWorks requires support of domain library including Modelica Standard Library (MSL) and user's special domain library. These libraries can be reused after they are loaded when system starts up. With the development of Modelica language and the technology of multi-domain physical system modeling and simulation, MSL and user's library become more and more complex and larger increasingly. As we know[7], MSL of Modelica2.1 have more than 3,100 defined classes. And now in Modelca2.2 this number has exceeded 4,300. This trend consequentially results in the following two problems:

1) Time cost becomes larger during the loading process of domain library. When system starts up, this process is executed automatically. If the library becomes very, very complex and large, the loading process may be unbearable.
2) Time cost becomes larger during the translating process of the main simulation class. For loading process does not construct document object model (DOM), translating a model needs building the structured information (i.e. DOM) of the reused classes from domain library at first.

Apparently, simulating a realistic complex model could consume a lot of time for translating the correlative classes from domain library ever built.

To resolve these problems, we put forward and implement the so-called domain library preprocessing technology in MWorks platform. The basic idea of this approach is that: We save all the document object model information of domain libraries through serializing binary DOM files only once when MWorks is set up or starts up for the first time. Afterwards, MWorks will load these DOM files every time during it starts up. Because the DOM file is only one binary file, loading it becomes very quickly. And also because the DOM file includes enough structured information of all the classes within domain library, DOM building processes of extends classes or component referenced classes of the current complex simulation class will be skipped. So through domain library preprocessing, simulating a complex model which reuses a lot of classes from the domain library will become very quickly even at the first time.

The next paragraphs of this paper will expatiate upon this approach in detail.

2 Traditional library loading and class translating processes

2.1 Traditional library loading

Traditional library loading process is a reading and pre-checking process of a series of mo text files according to the directory defined in system register table or MODELICAPATH environment variable. Usually, software based on Modelica loads the basic Modelica Standard Library including Modelica, ModelicaAdditions, MultiBody, etc. The traditional library loading process only performs the lexical and syntactical analyses, and then builds the tree structure of domain classes according to the mo file and directory structure, package and nested classes structure. Because the loading files are text format, its directory structure is complex and the process needs lexical and syntactical analyses, traditional library loading process is time-consuming. As our statistics, loading all the classes of Modelica2.1 needs more than 5 minutes through using lexical and syntactical analyzing tool ANTLR.

In addition, building tree control of domain library is also a waste-time process, for this process needs generate icons of the loading classes, which correspond to the images of tree nodes. The image of each tree node is created by drawing the icon annotation of the corresponding class. For this process involves creating and drawing geometric entities, as our statistics, building all the tree nodes of Modelica2.1 needs more than 8 seconds. Even if we adapt the method by saving the images to temporary bitmap files, this process still needs about 5 seconds.

2.2 Traditional class translating

As we mentioned above, loading process of domain library only pre-checks the classes but does not construct structured information which are necessary for the main class translating.

Traditional class translating process usually includes three steps: 1) Lexical and syntactical checking. This step usually depends on professional translating tool such as ANTLR. At the same time, this checking process will create abstract syntax tree (AST) of the main class. 2) Semantic resolving based on AST. It is to check types and resolve extends clauses, modifications (including re-declarations), outer or inner matches, connect clauses, and so on. 3) Equation system generating. The semantic resolving prepares necessary information for instantiation which is performed. The purpose of instantiation is mainly to generate equation system (continuous equations and discrete events) for the main class.

Because the main class may include quite a number of components and extends classes, translating it needs checking referenced classes of the components and extends classes, then generating their ASTs recursively. So if the main class includes a great many components or extends which are referenced from domain library, this checking process of the referenced classes could take long time.

3 Document object model (DOM) of Modelica class

As an immediate expression of program language, abstract syntax tree (AST) is adopted abroad to storage

structured information of class. But building AST of Modelica code is a very complicated task, for Modelica language has a complex syntax structure. Moreover, translating operation based on AST is more complex. For example, in Modelica language we can use the keyword "*final*" to constrain element or class being modified further, as the following code:

```
final class Volt
  String quantity = "Voltage";
  String unit = "Volt";
  String displayUnit = "V";
end Volt;
```

For all the class is defined as *final*, its property variables (quantity, unit, displayUnit) are also looked as *final*. Just to say they all can not be modified. The prefix deduction of AST needs traverse all nodes of the class *Volt* to define which elements need be set *final* property. Another more troublesome thing is that we must add nodes to AST or find particular node to modify its properties. So, we put forward and design the so-called ModelicaDOM, i.e. Modelica document object model, which is a type of container structure.

DOM (Document Object Model) is developed from XML (eXtensible Markup Language). DOM is defined formally as [8]: "Document Object Model of a type of document is a platform-independent and language-independent interface. It allows program or script to access or modify the content, structure and style of the document".

ModelicaDOM is a container in physical structure while stores the tree information of Modelica-based document in logical structure. We design the class ModelicaDOM and implement its construct function and other access functions which are supplied to access the inner data of its objects.

Relative to AST, ModelicaDOM has the following advantages:

(1) ModelicaDOM is an object-oriented expression style. Its nodes of container could be objects of a class or objects of derived classes of the class.

(2) Logical structure of the syntax tree with ModelicaDOM is simpler. For Modelica2.1, We can conveniently storage all its grammar information. According to grammar, a regular class can include information of seven types of elements: imports, extends, nested classes, components, equations, algorithms and modification information. So the definition of regular class should comprise container of this elements. Contrasting with AST, the number of classes which need be designed in ModelicaDOM hierarchy decreases about one third.

(3) Access operation of ModelicaDOM is simpler and more efficient. Finding some element or modifying data of some element is very convenient and quickly. For example, if we want to modify the data of a component, we can search and get the object pointer of the component class firstly, then modify the corresponding data through interfaces of the class. However in AST, these operations often need traverse the whole tree in order to find and modify the data.

So as a replacer of AST, *ModelicaDOM*, document object model of a defined class includes all information of the class in structured data format. DOM of a class can be built after the checking process of class. And it is the base of generating equation system of the class.

We designed the class hierarchy of ModelicaDOM according to Modelica semantics like fig. 1.

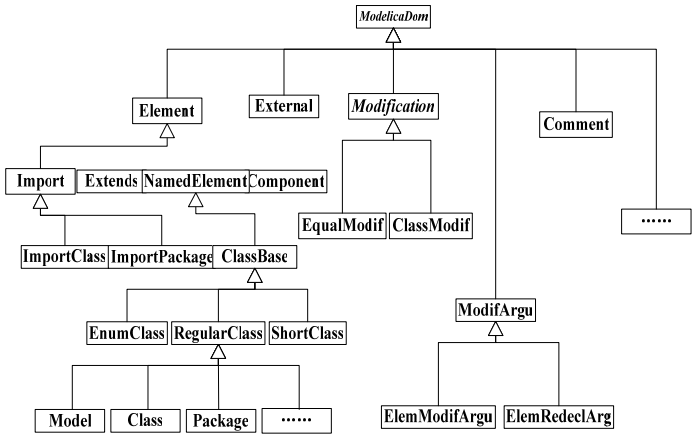

Fig.1 class hierarchy of ModelicaDOM

The *RegularClass* class in the hierarchy is one of the most important classes. *RegularClass* class has the following eight derived classes: Model, Class, Package, Block, Record, Type, Function and Connector classes. Thereinto, the "*RegularClass*" class is a common class which is usually used to define a simulation model. It contains about seven kinds of information: imports, extends, nested classes, components, equations, algorithms and modifications. In MWorks, "*RegularClass*" class is designed as following:

```
class RegularClass :
{
public:
  . . .
  //vector pointer of import classes
  vector<ImportClass*>*       pImpClasses;
  //vector pointer of import packages
  vector<ImportPackage*>*  pImpPackages;
  //vector pointer of extends classes
  vector<Extends*>*  pBases;
  //vector pointer of nested classes
  vector<ClassBase*>*  pNestedClasses;
  //vector pointer of components
  vector<Component*>*  pComponents;
  //vector pointer of annotations
  vector<Annotation**  pAnnotations;
```

```
    //container pointer of equations
    BehaviorContainer*  pEquaContainer;
    //container pointer of initial  equations
    BehaviorContainer*  pInitEquaContainer;
    //container pointer of algorithms
    BehaviorContainer*  pAlgoContainer;
    //container pointer of initial algorithms
    BehaviorContainer* pInitAlgoContainer;
    //pointer of modifications of this class
    ClassModif* pClassModif;
    . . .
};
```

But according to Modelica2.1, there are other classes (enum class and short class) excluding the regular class. So we design the class "*Classbase*" as the base class of these three classes, which defines all the same properties and same operations of these classes. And *Classbase* will be the base class of all node objects of the container of ModelicaDOM.

4 Preprocess of domain library in MWorks

When MWorks system starts up, it will execute the OnStartup() function automatically like the following code in C++:

```
//using MFC
void  CMainframe::OnStartup(vector<ClassBase*> & vClasses;)
{/*vClasses is the container of ModelicaDOM, i.e. vector of pointer of Classbase*/

    /*getting ModelicaPath, these paths were set by user*/
  vector <string> vPaths = GetModelicaLibPaths();
  //foreach path do
for(int i=0; i<vPath.size(); i++)
{
     string    sPath = vPath[i];
     //search DOM file from work directory
     string sDOMFile = GetDOMFile(sPath);
     //if the Dom file exists
     if(sDOMFile != "")
    {
    /*read the Dom file, construct vector of Classbase objects*/
    DOMSerilize(sDOMFile, vClasses, DOM_READ);
    }
    /*if DOM file does not exist, execute domain library preprocessing*/
  else {
   /*load mo files in sPath, and build vClasses roughly*/
   vClasses = LoadPath(sPath);
   /*rebuild vClasses again, set the pointer value of referenced classes*/
   RebuildDOM(vClasses);
   /*generate a DOM file name according to work dir*/
  string sDOMFile=GenerateDOMFileName(sPath);
   //save DOM information to sDOMFile
   DOMSerilize(sDOMFile,vClasses,DOM_WRITE);
  }
   //create icons of domain library tree nodes
   ……
}
```

From above, *GetModelicaLibPaths()* function gets paths of Modelica library which will be loaded. If there exists MODELICAPATH environment variable, the paths will be extracted from this variable. If not, system will search the registered table of Windows to get paths from the key "Modelica_Lib_Paths" which was set while MWorks installing.

The function *GetDOMFile(sPath)* will get a DOM file according to the const string *sPath* and the work directory of MWorks. For an example, if sPath is "C:\\MWorks\\Library\\Modelica" and the work directory is "C:\\MWorks\\work", this function will return "C:\\ MWorks\\work\\Modelica.DOM".

The code of creating icons of domain library tree nodes is leaved out. This process is just like the *vClasses* created. When the tree nodes is building according to the *vClasses*, it will search the icon images from a bmps file, if the file does not exists, it will generate an icon from the icon annotation of the class. At the same time saving the icon image to a bmps file. For all icons are saved to one bmps file only if system starts up once, the icons of domain tree nodes will be created very quickly through reading the image from the bmps file, without creating images through constructing geometric entities and drawing them.

From the code above, preprocess of domain library includes three stages: library loading, ModelicaDOM building and serializing. This process is usually performed when MWorks starts up or software is installed. It can also be performed via loading library command. It is performed only once, and results in generating a DOM file.

4.1 Loading of domain library

Like 2.1, MWorks loads all mo files including structured entities and non-structured entities according to the specified library directory through executing the function *LoadPath*(). At the same time, MWorks performs lexical and syntactical analysis for the loaded files.

In addition, MWorks will build the DOM information roughly when checking, that is to say, it build all the nodes of the DOM container and set their main information. For there could exist inner reference relationships among the classes in domain library, the pointer of referenced classes of a class could not be all obtained when the class is constructing. Only after all the classes are constructed, these pointers information can be obtained.

4.2 Rebuilding of DOM

The *LoadPath*() function only created the DOM structure roughly, its detailed data are obtained through *RebuildDOM*() function after loading process.

As mentioned above, DOM of a class includes all structured information of the class. DOM of the domain library contains all structured information of all classes defined in the domain library. In MWorks, they are linked as an object-oriented container structure like the following fig.2 shows.

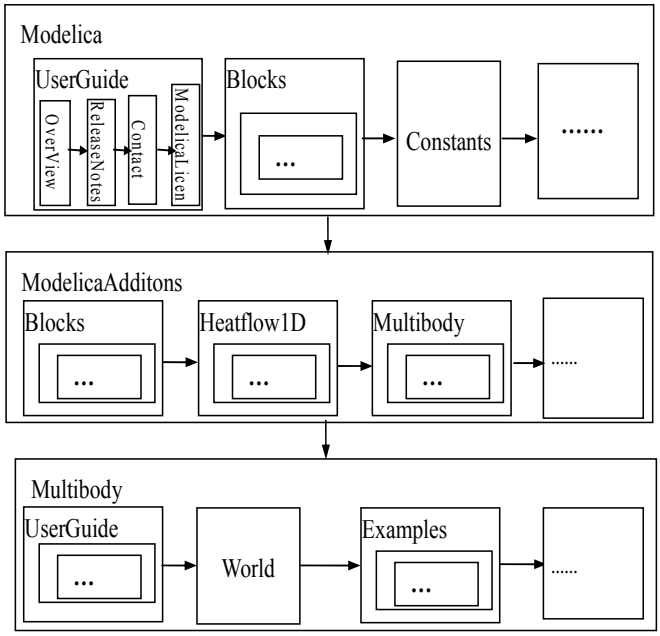

Fig.2 ModelicaDOM of domain library (Modelica2.1)

The container of fig.2 expresses the tree structure of domain classes like fig.3 shows:

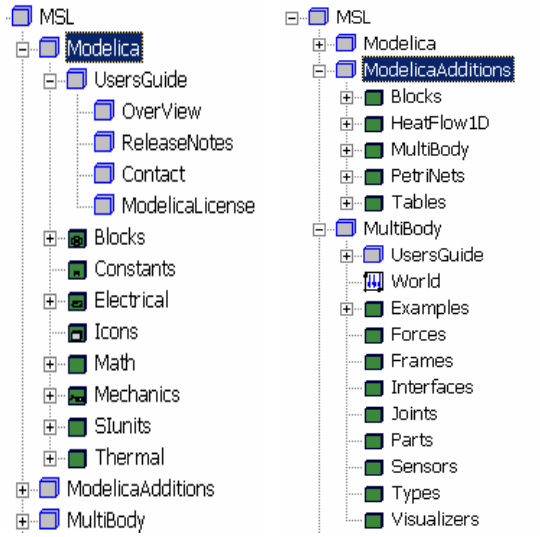

Fig.3 tree structure of domain library(Modelica2.1)

4.3 Serializing of DOM - Saving

Serializing of DOM involves two aspects: saving DOM information into DOM files and loading DOM files to construct DOM data structure. And the former is the main task of preprocessing of domain library. This is realized by the function *DOMSerialize*() whose third parameter *mode* is set DOM_WRITE.

After the DOM of one path in domain library has been constructed, they can be serialized into one DOM binary file according to DOM data. This process can be described as the following pseudo-code:

```
function
    DOMSerialize(vector<ClassBase*>vModels,
    string sDOMFileName, int mode =
    DOM_WRITE)
{
    if(mode == DOM_WRITE)
    {
        ofstream fs(sDOMFileName, ios::binery);
        //write the number of saved classes
        fs << vModels->size();
        //write data of each class
        foreach(pModel in vModels) {
        //write the type of class
        fs << pModel->GetType();
        //write data of class in detail
        pModel->WriteToDOMFile(fs);
    }
    else {//DOM_READ
        ...
    }
}
```

The function *WriteToDOMFile*() is virtual function of *ClassBase* class, which is implemented in the same functions of its derived classes *RegularClass*, *ShortClass* and *EnumClass*. From the class hierarchy of ModelicaDOM, this function of *RegularClass* is implemented through its derived classes Model, Class, Package, Block, Record, Type and Connector. We give the function code of *RegularClass* as an example:

```
void RegularClass::WriteToDOMFile(ofstream &ofs)
{
   //save  import classes
   ofs << pImpClasses->size()；
   for(int i=0; i<pImpClasses->size(); i++)
   {
      ofs << (*pImpClasses)[i]->GetFullName();
   }
   .. .. ..
   //save  extends classes
   ofs << pBases ->size();
   for(int i=0; i< pBases ->size(); i++)
   {
     (* pBases)[i]->Write(ofs);
   }
   //save nested classes recursively
   ofs <<  pNestedClasses->size();
   for(int i=0; i<pNestedClasses->size(); i++)
   {
      pNestedClasses[i]->WriteToDOMFile(ofs);
   }
   //save  components
   ofs << pComponents ->size();
   for(int i=0; i< pComponents ->size(); i++)
   {
     (*pComponents)[i]->Write(ofs);
   }
   .. .. ..
}
```

5 Library loading, and class translating in MWorks

5.1 Library loading in MWorks

As another aspect of serializing, loading is the reverse operation of saving which is implemented through the function *DOMSerialize()* with the third parameter mode being DOM_READ. In MWorks, library loading process can be described as the following pseudo-code:

```
function   DOMSerialize(vector<ClassBase*>vModels,
      string sDOMFileName, int mode = DOM_WRITE)
{
   if(mode == DOM_WRITE)
   {
      ……
   }
   else { //DOM_READ
      ifstream ifs(sDOMFileName, ios::binery);
      //read the number of classes ever saved
      ifs >> num;
   for( i =0; i<num; i++)
   {
        //read type of class
          ifs >> type;
          //Create object of the class
          ClassBase * pModel ;
          if( type == “shortclass”)
             pModel = new ShortClass();
         else if(type == “enumclass”)
             pModel = new EnumClass();
         else if(type == “class”)
             pModel = new Class();
        else if(type == “model”)
             pModel = new Model();
        ……
        pModel->ReadFromDOMFile(ifs);
        //append to vModels
          vModels->append(pModel);
      }//end for
   }//end of else
}
```

The function *ReadFromDOMFile()* is executed in the same order with saving like the following code:

```
void RegularClass::ReadFromDOMFile(ifstream &ifs)
{
   //read import classes
   int nImpSize;
   ifs >> nImpSize;
   for(int i=0; i<nImpSize; i++)
   {
      string sImpName;
      ifs >> sImpName;
      ImportClass*pImpClass=
            new ImportClass(sImpName);
      pImpClasses->push_back(pImpClass);
   }
   .. .. ..
   //read extends classes
   int nExtendsSize;
   ifs >> nExtendsSize;
   for(int i=0; i< nExtendsSize; i++)
```

```
  {
    Extends * pExtends = new Extends();
    pExtends ->Read(ifs);
   pBases->push_back(pExtends);
  }
  //read nested classes recursively
  int nNestedClassSize;
  ifs >> nNestedClassSize;
  for(int i=0; i<nNestedClassSize; i++)
  {
    ClassBase * pClass = new ClassBase();
    pClass->ReadFromDOMFile(ifs);
   pNestedClasses->push_back(pClass);
  }
  //read components
  int nComSize;
  ifs >> nComSize;
  for(int i=0; i< nComSize; i++)
  {
    Component * pComp = new Component();
    pComp->Read(ifs);
   pComponents->push_back(pComp);
  }
  .. .. ..
}
```

From these pseudo-codes we can see, MWorks lookups the DOM file which contains the structured information of domain library from the installed directory first. Only if this process fails (DOM files does not exist), it will load domain library and execute preprocessing. If DOM file exists, MWorks will perform serializing process: loading but not parsing the DOM file, then rebuilding the DOM structured information of domain library in the memory.

5.2 Class translating process in MWorks

In MWorks, translating process of main class can be described as the following pseudo-code:

```
void RegularClass::Translating(EquationSystem & es)
{
   //generate equations of this class
   //container pointer of equations
  pEquaContainer->GenerateEquSys(es);
  //container pointer of initial equations
  pInitEquaContainer->GenerateEquSys(es);
  //container pointer of algorithms
  pAlgoContainer->GenerateEquSys(es);
  //container pointer of initial algorithms
  pInitAlgoContainer->GenerateEquSys(es);
  //pointer of modifications of this class
  pClassModif->GenerateEquSys(es);

  //generate equations of extends classes recursively
  for(int i=0; i< pBases ->size(); i++)
  {
    string sClassName=(*pBases)[i]->GetFullName();
    ClassBase * pClass = GetClass(sClassName);
    if (pClass == null) {//does not build
      pClass = BuildDOM(sClassName);
    }
    pClass->Translating();
  }
  //neglect nested classes
  //generate equations of components
  for(int i=0; i< pComponents ->size(); i++)
  {
     Component *pComp = (*pComponents)[i];
     string sCompClass = pComp->GetClassName();
     ClassBase * pClass = GetClass(sCompClass);
     if(pClass == null) {
       pClass = BuildDOM(sCompClass);
     }
     pClass->Translating();
     pComp->GenerateEquSys(es);
  }
  ……
}
```

The function *BuildDOM*() involves getting the path from full-name of the class, loading it with *LoadPath*() function and rebuilding DOM with *Rebuilding*() function.

As we know, first step of translating the main class is to build DOM of the class. And building the DOM of main model will lookup or construct DOM of its extends classes and component classes which usually referenced from the domain library. Because the domain library has been loaded when system starts up and DOM of the library has been constructed, this process only needs looking-up of DOM referenced classes. And this process only needs getting pointers to the DOM of domain library.

5.3 Generating Fulltext of Class from DOM

As a Modelica-based platform, MWorks has five views which express five aspects of the class respectively: *Fultext* view, *Icon* view, *Diagram* view, *HTML* view and *Simulation* view. The *Icon* view and the *HTML* view express icon and HTML information of the class and their information is obtained from the *pAnnotations* variable of the class and the base classes recursively. The *Diagram* view expresses diagram entities, components blocks and connect among components of the class, so its information comes from the *pAnnotation* variable of the main class and its base classes recursively, *pComponents* and connect part of *pEquaContainter* of the main class. The *Fulltext* view is mo text expression of the main class. Because we have adopted domain library preprocessing, the mo text information of the classes

in the domain library was lost. So if we want to show the full-text of these classes, the mo text of each class must be reconstructed from the DOM structure.

Generating process of the fulltext of a class is similar as the saving process from DOM information to DOM file. And the distinguish from saving is that it prints the DOM information to its original mo text string while saving process saves all the property data of the class to a binary file. As an example, the import class information *pImpClasses* converts to mo text like the following code:

```
void RegularClass::ToString(string & sText)
{
  //print pImportClass to string
  for(int i=0; i< pImpClasses->size(); i++)
  {
    sText += "import ";
    sText += pImpClasses[i].GetFullName();
    sText += ";\n";
  }
  ……
}
```

6 Conclusions

We have implemented a modern IDE for modeling and simulation of multi-domain physical systems based on Modelica - MWorks. It has a larruping feature of domain library preprocessing, which improves loading speed of library and translating speed of the simulation model rapidly. But from other point of view, domain library preprocess needs much larger memory space. In other words, we sacrifice space complexity to gain time saving. The following is time and space consuming contrast:

Time contrasting: for Modelica2.1, using a computer with its CPU being 2.4Gmps, memory being 512M, loading mo text file needs more then 300s, while if we adopt preprocessing, loading the DOM file only needs 6s.

Space contrasting: with the same computer, loading without building DOM structure only needs 11M, while through preprocessing the occupied memory will reach to or exceed 100M.

In addition, for the DOM file saves the structured information but not in mo text format, classes in the library can be encrypted easily through the pre-processing.

7 Acknowledgement

The paper was supported by project of Chinese National Science Foundation Committee (60574053), Chinese 863 high tech project (2003AA001031) and Chinese 973 project (2003CB716207).

References

[1] Modelica, http://www.modelica.org/, 2006.

[2] Dynasim. Dymola, http://www.dynasim.se/, 2006

[3] MathCore. MathModelica, http://www.mathcore.se/, 2006

[4] OpenModelica Fritzson, P., et al. *The Open Source Modelica Project*. in *Proceedings of The 2th International Modelica Conference*, 18-19 March, 2002. Munich, Germany.

[5] C. Nytsch-Geusen et. al., Fraunhofer Institutes, Germany: MOSILAB: Development of a Modelica based generic simulation tool supporting model structural dynamics in *Proceedings of The 4th International Modelica Conference*, 7-8 March, 2005. Hamburg-Harburg, Germany.

[6] M. Najafi, S. Furic, R. Nikoukhah, Imagine; INRIA-Rocquencourt, France: SCICOS: a general purpose modeling and simulation environment in *Proceedings of The 4th International Modelica Conference*, 7-8 March, 2005. Hamburg-Harburg, Germany.

[7] Peter *Fritzson*, *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*, Wiley-IEEE Press, 2004

[8] Document Object Model (DOM). http://www.w3. org/DOM/, 2006