CD++ Repository: An Internet Based Searchable Database of DEVS Models and Their Experimental Frames

Rachid Chreyh, Gabriel Wainer
Department of Systems and Computer Engineering
Carleton University
Ottawa, ON K1S-5B6 Canada
rchreyh@connect.carleton.ca, gwainer@sce.carleton.ca

Keywords: DEVS, CD++, Model Libraries, Model Re-use, Experimental Frames

Abstract

The development of simulation models for complex systems can be difficult and time consuming. This makes the ability to re-use modelling components of high value. To be able to re-use modeling components it is important to know the context within which a given component was developed. Experimental Frames are useful for capturing this context. We present the CD++ Repository – an internet based searchable database of re-usable CD++ DEVS models and their Experimental Frames. CD++ Repository facilitates the re-use of models and allows users in different geographical locations to collaborate in building complex models. The database is built as a hierarchy of the stored atomic and coupled models, thus eliminating repetition. One of the key features is that along with the storage of the atomic and coupled models, it stores Experimental Frames for each model, which allows users to easily determine the context for which a given model applies.

1. INTRODUCTION

Modeling of complex systems is a difficult and time- consuming endeavor. Applying the concept of component reuse to the modeling process greatly reduces the time and effort required to create new and more complex models. DEVS's hierarchical and modular approach to model creation allows for models to be independently tested and reused thus enhancing reliability, maintainability and reducing the time and effort required for model development and testing [1]. The creation of DEVS models, however, involves examining the source system through an "Experimental Frame" (EF) which defines the conditions under which a system is observed or experimented with [1]. A DEVS model thus aims to approximate the behaviour of the source system within the parameters set by the EF.

Similarly, the experiments used to verify the validity of a given DEVS model can be difficult and time consuming to develop. The ability to re-use the experiments developed for one DEVS model during the testing of other similar DEVS models would be of great benefit to the modellers. For example, if a model of a traditional keyboard exists and there is a number of experiments to test all of the different combinations of button presses, these experiments could be reused to test a new model of a touch-screen keyboard.

For modellers to make use of the re-usable DEVS components we propose a method to access and search a repository of components. For it to be useful, such a repository of DEVS modeling components must contain, for each component, a description of its EF so that users are able to determine the context within which a given component is valid. In addition, storing the experiments for each stored model would be of great benefit to the modellers. Finally, the increasing need for teams of modellers located at different geographical locations to work with the same modelling components means that such a repository would benefit from being accessible over the internet.

This work introduces such a models library: the CD++ Repository which is based on the CD++ Builder Toolkit [2]. CD++ Repository is composed of a web-based database server and a client application built as an addition to the CD++ Builder toolkit. The CD++ Repository's database is capable of storing Atomic and Coupled CD++ DEVS models in addition to CD++ Cell-DEVS models [3]. The database is built as a hierarchy of the stored atomic and coupled models, thus eliminating any needless repetition. In addition, for each stored model, the database stores EF information, which helps users determine the validity of a model within different contexts, and any number of experiments associated with the model. The CD++ Repository's client application is embedded within the CD++ Builder Toolkit [2] and it enables the user to search for, store, and retrieve DEVS and Cell-DEVS models and their EFs directly from the CD++ modelling and simulation environment. Finally the CD++ Repository Client connects to the CD++ Repository Database over an internet connection, thus allowing the access of the library of models and EFs from any geographical location.

2. EXPERIMENTAL FRAMEWORKS

The Experimental Frame (EF) concept has been introduced to capture the set of circumstances under which a real system is to be observed, or is to be subjected to experimentation [1, 5]. This makes the meaning of EF ambiguous in that it could mean different things within the M&S process (experimentation, data collection, modelling, simulation). In [6] the idea of a framework that takes into account the different meanings of an EF was introduced. These ideas were formalized by the Context-Frame-Experimentor framework introduced by Traore and Muzy [7]. This framework presents a clear distinction between the three levels of abstraction, namely: the context through which a real system is being studied, the specification of this context as an EF, and the implementation of this EF to execute on a simulator.

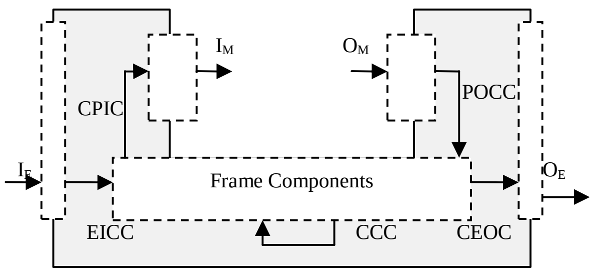


Figure 1. Experimental Frame Definition [7]

Using this framework of the EF, presented in Figure 1, one can formally describe the context by the EF. One can think of the EF as a circuit board with input and output ports into which a source system or a model can be 'pluged-in'. The EF itself can be a complex combination of components with their own interconnections and inputs and outputs. Then an EF can be described as [7]:

$$\mathbf{EF = <T , I_M , I_E , O_M , O_E , \Omega_M , \Omega_E , \Omega_C , D , \{C_d , d \varepsilon D\}, CPIC, EICC, POCC, CEOC, CCC>}$$

T is a time base, $\mathbf{I_M}$ is the set of Frame-to-Model input variables, the plug-in input set, $\mathbf{I_E}$ is the set of Frame input variables, the control input set, $\mathbf{O_M}$ is the set of Model -to-Frame output variables, the plug-in output set. $\mathbf{O_E}$ is a set of Frame output variables, the summary set, $\mathbf{\Omega_M}$ is the set of admissible input segments for the plug-in component, the plug-in input constraints set, $\mathbf{\Omega_E}$ is the set of admissible input segments for the experimentation control, the control input constraints set, $\mathbf{\Omega_C}$ is the set of admissible output segments expected from the plug-in component, the plug-in output constraints set. **D** is a set of component names, the control components set, $\mathbf{C_d}$ is a model for each d ε D, **CPIC** is the Control-to-Plugin-Input coupling. Finally, **EICC** is the External-Input-to-Control coupling, **POCC** is the Plugin-Output-to-Control coupling, **CEOC** is the Control-to-External-Output coupling, **CCC** is the Control-to-Control coupling.

3. CD++ REPOSITORY ARCHITECTURE

The CD++ Repository is comprised a server and a client component. The server contains the Central Database of Models and their EFs, hosted on a computer connected to the Internet. The client component is the CD++ Repository application accessible from inside CD++ Builder (an Eclipse plug-in).

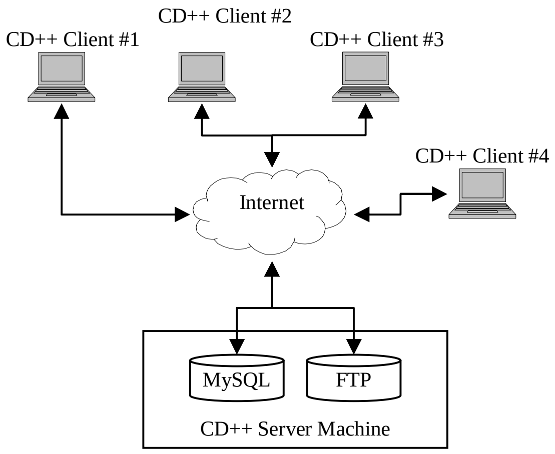


Figure 2. Architecture Diagram

CD++ Repository uses a MySQL Database and an FTP server to access files related to the stored entities. The CD++ Repository software residing on a client machine can connect to these servers in order to upload/download information: Atomic and Coupled Models, EFs (including EF Inputs and Outputs, and Ranges), Experiments, and Experimental Results. The following sections will address the following questions:

1. For a given model, EF, Experiment, or Experiment Result, what kind of information is stored in the database?
2. How are the relationships between the models (Atomic and Coupled) handled?
3. How is the relationship between the Models, EFs, the Experiments, and Experimental Results handled?

3.1. Atomic and Coupled DEVS Models

CD++ Builder uses three files to describe any Atomic Model. The first two are the C++ class files (*.cpp* and.*.h*) derived from the abstract C++ class "*Atomic*"; they describe the behaviour of the Atomic model being developed. Coupled and Cell-DEVS Models are described on a model definition file (*.ma)* using a built-in specification language.

The following information (referred from now on as the *Model Data*) has been identified and collected:

1. *Model Name*: a unique name generated by the CD++ repository software.
2. *Domain*: the domain under which the model can be categorized. (i.e. Telecommunications Equipment, Urban Traffic, etc.)
3. *Title*: a title for the model.
4. *Acronym*: an acronym for the model.

5. *Brief Description*: a short paragraph describing the model's general characteristics.
6. *Key Words*: They are associated with the model, and can be useful when doing a search of the repository.
7. *Developer Name*.
8. *Creation Date*.

When Coupled DEVS models are stored their 'relationships' with other models have to also be stored in the database. To this end, the CD++ Repository maintains the hierarchical construction of the Coupled models even while they are stored. This is done by storing the Coupled model as a separate entity (with its own *Model Data* and (.ma) file) and linking it to its tree of child models in the repository. For example, for the model in Figure 3, each of these models are stored separately in the repository, with Model A linked to Models X and Y, and model B linked to models A and Z.

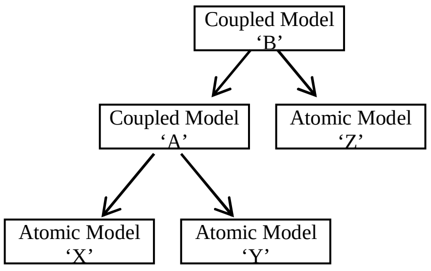


Figure 3. Stored Coupled Models

3.2. Experimental Frames

An EF for a given model contains some *Experimental Frame Data* (analogous to the *Model Data*), a set of Experiments that can be run on the Model, and other information. The *Experimental Frame Data* is comprised of textual descriptions that can be used to specify the context within which a given model is valid including:

1. *Objectives* for which the model was built.
2. *Assumptions* made by the model designer when designing this model.
3. *Constraints* within which the model was designed to operate.

More detailed EF information is included to deal with the formal definition of the EF. This information captures all of the inputs and outputs of the EF and the valid range of values for each input or output. This information is stored in such a way as to eliminate any repetition.

3.3. Experiments

Experiments are stored separately and linked to the EF of the model to which they apply. A given experiment may apply to more than one model and thus may be linked to more than one EF. Like the Models and EFs, each experiment also has descriptive data stored with it which will be referred to as the Experiment Data. The Experiment Data is comprised of the following pieces of information:

1. *Experiment Name*: a unique name generated by the CD++ repository software.
2. *Title*.
3. *Brief Description*: a short paragraph describing the experiment in general.
4. *Objectives* for which the experiment was created.
5. *Assumptions* made by the designer of the experiment.
6. *Constraints* within which the experiment was designed to operate.
7. *Developer Name*.
8. *Creation Date*.

It is important here to explain the difference between the objectives, assumptions and constrains of the experiment as opposed to the EF. For the EF these pieces of information refer to the model and the characteristics of the model itself, however for the experiment they refer to the experiment itself and thus are usually a sub-set of those in the EF.

3.4. Experimental Results

CD++ Repository also keeps Experimental Results for a given Experiment on a given Model. Since Experiments may apply to more than one Model, the information is not directly linked to the Experiment. Instead, it is linked to the Model on which it was performed. The Experimental Results information is comprised of the following:

1. *Success*: A Boolean value to indicate success or failure of the experiment
2. *Description Document*: A document containing any comments related to the performance or the results.
3. *Log file*: The (.log) file generated by CD++ for this run of the experiment. This file contains many details of the CD++ simulator's actions during simulation.
4. *Output file*: The (.out) file generated by CD++ for this run of the experiment, containing the values presented at the outputs of the model under test and the time the outputs took that value.

4. THE CD++ REPOSITORY CLIENT

The CD++ Repository Client was built as part of the CD++ Builder plug-in in Eclipse. When the user activates the CD++ Repository application within Eclipse they are presented with one of the following options:

a) Uploading Models and Experiments

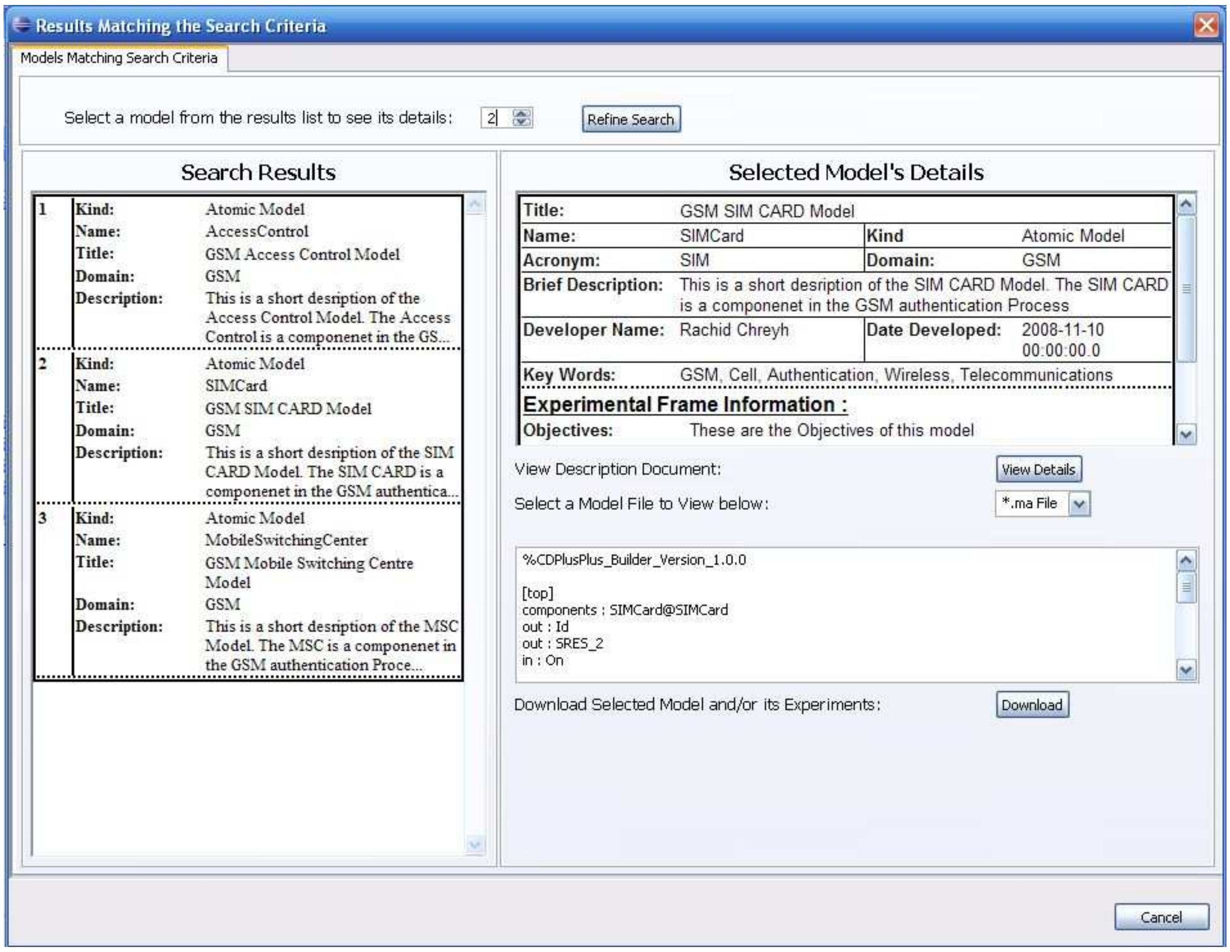
When uploading models to the database the Repository Software will automatically:

1. Detect what kind of Model this is (Atomic, Coupled, Cell-DEVS).
2. For Coupled models, detect all the sub-models and construct the full hierarchy under the parent model.
3. Establish a name for this model and all sub-models.
4. Detect conflicts with models that already exist in the repository (and handle the conflict appropriately).

5. For Coupled models: detect whether any of the child models already exists in the Repository, and if they do handle the conflicts appropriately.
6. For Coupled models: construct a list of all the models that do not already exist and for which data needs to be collected from the user.
7. Finally, collect the required Model Data, EF Data, and the files for the model and child models (if applicable) and upload all of this information to the Repository.

b) Search and Download of Models and Experiments

When the searches for a model or experiment they are presented with a simple search dialogue window which allows the user to search for a model based on some of the Model Data (or Experiment Data). After performing the search, the CD++ Repository presents the search results to the user. Figure 4 shows a screenshot of the model search results screen (the experiment search is similar except that it contains experiments and experiment related information).


Results Matching the Search Criteria
Models Matching Search Criteria
Select a model from the results list to see its details: 2
Refine Search
Search Results
Selected Model's Details
View Description Document:
View Details
Select a Model File to View below:
*.ma File
Download Selected Model and/or its Experiments:
Download
Cancel

1	**Kind:**	Atomic Model
	Name:	AccessControl
	Title:	GSM Access Control Model
	Domain:	GSM
	Description:	This is a short desription of the Access Control Model. The Access Control is a componenet in the GS...
2	**Kind:**	Atomic Model
	Name:	SIMCard
	Title:	GSM SIM CARD Model
	Domain:	GSM
	Description:	This is a short desription of the SIM CARD Model. The SIM CARD is a componenet in the GSM authentica...
3	**Kind:**	Atomic Model
	Name:	MobileSwitchingCenter
	Title:	GSM Mobile Switching Centre Model
	Domain:	GSM
	Description:	This is a short desription of the MSC Model. The MSC is a componenet in the GSM authentication Proce...

Title:	GSM SIM CARD Model		
Name:	SIMCard	**Kind**	Atomic Model
Acronym:	SIM	**Domain:**	GSM
Brief Description:	This is a short desription of the SIM CARD Model. The SIM CARD is a componenet in the GSM authentication Process		
Developer Name:	Rachid Chreyh	**Date Developed:**	2008-11-10 00:00:00.0
Key Words:	GSM, Cell, Authentication, Wireless, Telecommunications		

Experimental Frame Information :

Objectives:	These are the Objectives of this model

```
%CDPlusPlus_Builder_Version_1.0.0

[top]
components : SIMCard@SIMCard
out : Id
out : SRES_2
in : On
```

Figure 4. Search Results Screen Shot

The Search Results window is divided into three main areas. The top area contains a drop down list (to select one of the search results) and a *refine search* button. The bottom left area is the *Search Results Pane*, which contains a list of entries representing a model that matched the search criteria. Each of these entry contains some general information about the model that it represents. Finally, the bottom right area is the *Details Pane*, containing all of the information concerning the selected model. It includes an area in which all of the Model Data and Experimental Frame Data for the selected model is displayed, including a listing all of the experiments for the selected model and the experimental results for each experiment (here, some of the information for the model *SimCard*). The *View Details* button enables the user to view the detailed description document for the selected model. Finally, a drop down list and a text box enables the user to view the text contained in any of the selected model's files. The user can select to display the text contained in the files by selecting the appropriate file type from the drop down list. When the user has found a model that they want to download, they can proceed.

c) Modifying Models and Experiments

A user can choose to modify a model or experiment that already exists in the database. This feature allows the user to add/remove experiments to/from the EF of a given model. It also allows the use to modify any of the following information: Model Data, Experimental Frame Data, or Experiment Data. However this feature cannot allow a user to modify the actual model files stored in the repository.

5. CD++ REPOSITORY ARCHITECTURE

CD++ Repository uses a client-server architecture with the client application being the CD++ Repository portion of the CD++ Builder plug-in, and the server being the database server running on a remote machine where all of the data objects are stored. CD++ Repository is a "thick client" application (all of the logic is carried out on the client machine while the interaction with the server is solely to send and retrieve data to and from the database).

CD++ Repository's client application is made up of two layers, the *presentation* and the *persistence* layer. Presentation is concerned with interactions with the user through displaying data retrieved from the database and collecting data to send to the database. It uses the persistence layer to search the database, retrieve data from the database, make decisions about what to display, send data to the database, and detect conflicts. One final important part of the architecture are the Business Objects, which encapsulate the business data and their behaviours. They are used to exchange data between the presentation and persistence layers and their behaviours are used by the presentation layer to perform some processing on the data in the objects.

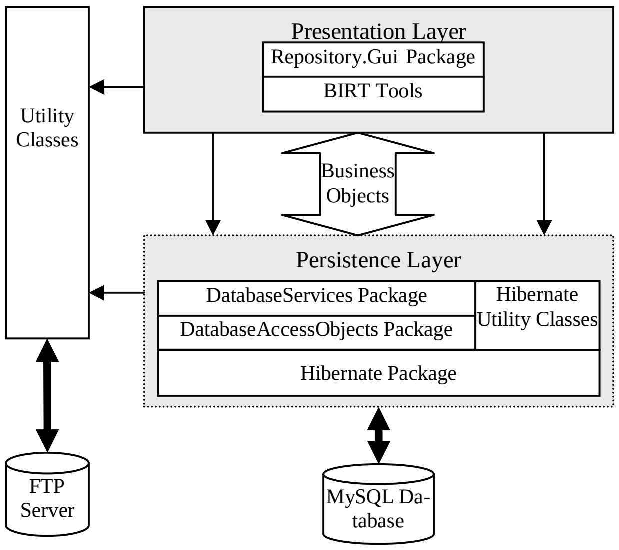


Figure 5. Software Architecture

Figure 5 gives a high-level view of the CD++ Repository's client application. As can be seen in the diagram, a major part of the persistence layer is the Hibernate Java Package. This is the object relational mapping tool used by the CD++ repository to map the Business Objects to their corresponding Relational Tables in the database [8]. The presentation layer makes use of the BIRT [9] (Business Intelligence reporting Tools) package, an Eclipse plug-in that enables the design and the runtime generation of reports and it is used in the display of search results to the user.

5.1. The Business Objects

The class diagram Figure 6 shows the most important Business Object (BO) Classes of the CD++ Repository software. The central class is the *Model* class, which represents all of the information that relate to Atomic/Coupled models. In addition, it contains a set of *ExperimentalResults* objects and a single *ExperimentalFrame* object. This reflects the fact that each model can have many Experimental Results and only one EF. The *ExperimentalFrame* class contains a set of *Experiment* objects representing the experiments that can belong to a given model. *AtomicModel* and *CoupledModel* classes extend the base *Model* class. Finally the *Experiment* class has two *AtomicModel* class objects and two *CoupledModel* class Objects. These represent the possibility that an Experiment can be model based (as opposed to event based) in which case it would have a model as a 'generator' and another model as a 'transducer'.

The class diagram in Figure 6 does not show the methods for each class. In general each class attribute has getter and setter methods, and other methods common to all of the Business Objects (i.e., *equals* and *hashCode*, defined by the Java language for all Java objects). These methods are overridden to provide proper object identity for the BO's. Some classes have other methods that perform simple operations related to each class:

a) Model:

- *getExpFilename*: given the name of a particular experiment, it finds the matching experiment in the set of experiments for the current model.
- *getMatchingExperiment*: finds a matching experiment in the set of experiments for the current model, and returns the experiment object.
- *createZipFile* as in Model before creates a (.zip) file containing all of the files related to the current model and returns the name and path of the created (.zip) file.

b) CoupledModel:

- *findCoupledSubModelByName*: finds the coupled model in the tree of models.
- *findParentofCoupledSubModelByName*: finds the coupled model in the tree, and returns the parent model of the coupled model found.
- *retrieveAllAtomicSubModels*: returns a set of all of the Atomic model objects for this coupled model.

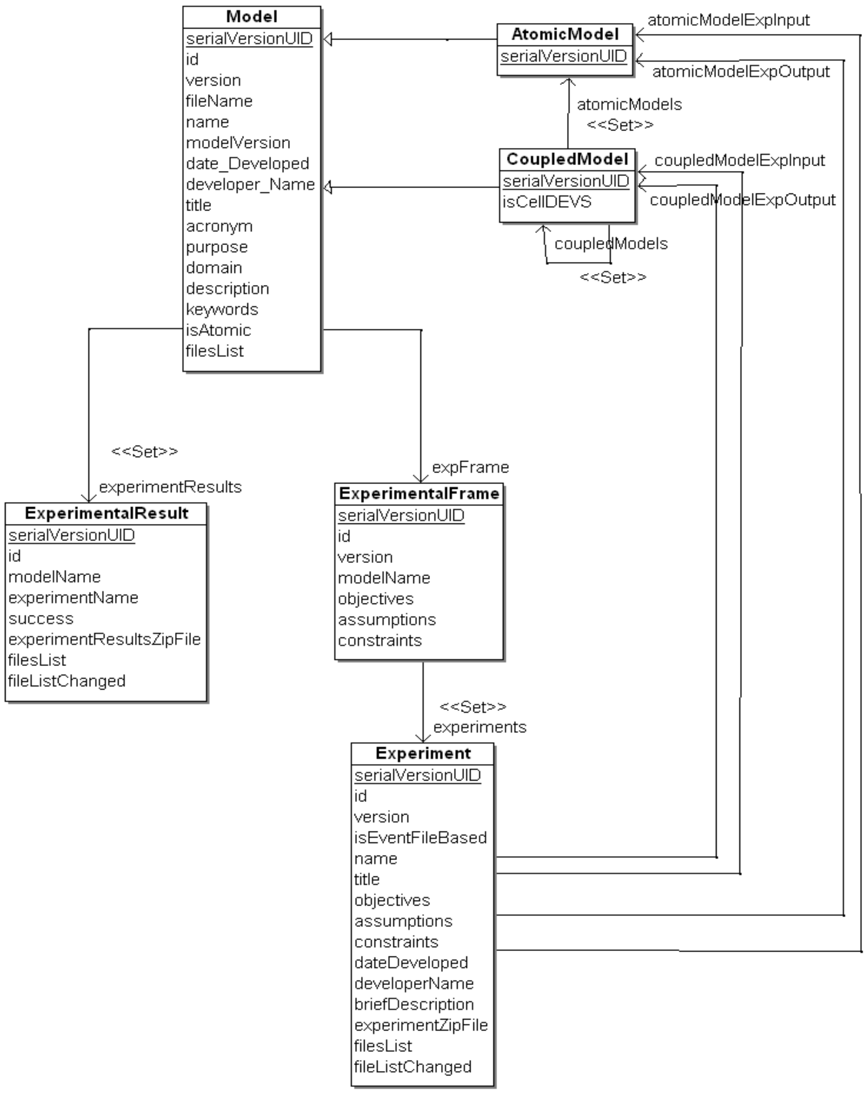


Figure 6. CD++ Repository Business Objects

- *retrieveAllCoupledSubModels*: returns a set of all of the Coupled model objects for this coupled model.
- *sameStructure*: checkes if two coupled models have the same structure.
- *findChildCoupledModelByName*: finds the immediate child of this model that is a coupled model.
- *getAllFileNames*: returns a list of the names of all of the (.zip) files for all of the models in tree under this model.
- *updateModelInfo*: it updates selected attributes of the current coupled model.

c) Experiment:

- *genNextExpName*: it creates a unique name to be used for the next Experiment object to be created.
- *createZipFile*: creates a (.zip) file containing all of the files related to the current experiment.

d) ExperimentalResults:

- *createZipFile*: creates a (.zip) file containing all of the files related to the current ExperimentalResults

5.2. Hibernate and Object-Relational Mapping

An important question to a database application like the CD++ Repository is how to manage the application’s persistent data. The Business Objects described in the previous section are used to hold this persistent data during the life of the application using MySQL (a Relational Database. While the java application uses an object-oriented representation of the business entities (the Business Objects), the relational database uses a tabular representation of the same business entities (the database tables). The relational model and object–oriented model are two fundamentally different models of data representation [8]. This problem has been researched thoroughly and there are numerous attempts at solutions. We used Hibernate, which uses Object/Relational Mapping (ORM) techniques to solve the mismatch [8].

Hibernate uses XML (*Hibernate mapping files*) to map Java business objects into SQL tables. Each class that needs to be persistent must have a Hibernate mapping file that defines the database table that the class maps to and that maps the properties of the class to the appropriate columns or tables in the database. The tables generated for the CD++ repository are shown in Figure 7.

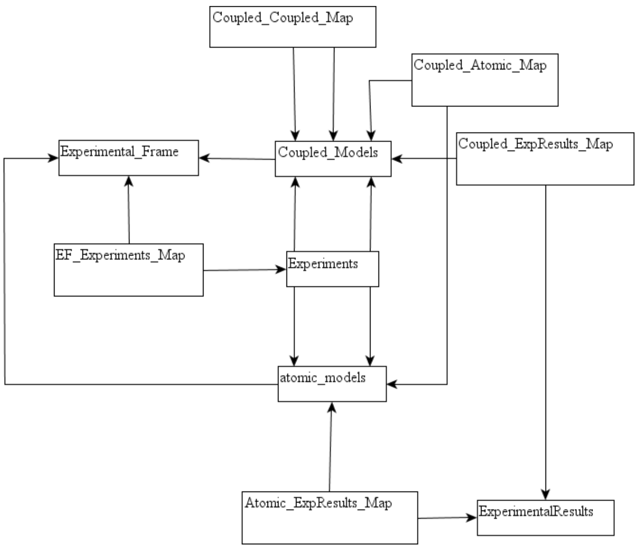


Figure 7. CD++ Repository Relational Tables

For the most part the database tables have similar names to the business objects to which they map. The rest of the database tables in Figure 7 are *join tables* that represent the interconnections between the objects. *Coupled_ Atomic_Map* and *Coupled_Coupled_Map* tables represent the fact that a Coupled model can have Atomic and Coupled models as children. *Coupled_ExpResults_Map* and *Atomic_ExpResults_Map* tables represent the fact that each atomic and coupled model can be associated to multiple experimental results. *EF_Experiments_Map* represents the fact that an EF can be associated with multiple experiments.

5.3. Persistence Layer Java Packages

Hibernate sits at the interface with the database; as such any of the CD++ Repository code that needs to interact with the database is required to interface through Hibernate. There are, however, many details that one should keep in mind to be able to use Hibernate correctly in a Java application: the management of Hibernate Sessions, Hibernate Transactions, Transient, Persistent and Detached Objects. CD++ Repository's software architecture hides most of the details that deal with the use of Hibernate inside the Persistence Layer. Therefore a few packages and utilities have been built to encapsulate the database services required by the Presentation layer such that Presentation layer code is not bogged down with Hibernate programming details.

The *HibernateUtil* Class is the utility used to start Hibernate. It contains a static SessionFactory attribute and a static getter function for it. An initialization method (*InitializeSessionFactory*) instantiates and configures the Hibernate SessionFactory by loading configuration information from the Hibernate configuration file as well as using the CD++ Repository preferences page to get information required to connect to the appropriate SQL server and database. Other utility methods that are made available to the Presentation Layer by this class are:

- *attachToSession*: attaches an object to a session. This is useful in situations where the object contains a graph of other objects whose values need to be accessed from the database.
- *commitSessionTx*: Used to end the transaction and close the session started by attachToSession.
- *mergeObject*: opens a session, and updates the database with the detached object's information.

RepositorySearchUtil provides the following methods to inspect the repository with respect to a given Model or Model name:

- *isInConflictWithRepo*: determines whether a passed in Coupled model has the same structure as the matching (by name) coupled model in the repository.
- *modelNameInRepository*: determines if the passed in model name already exists in the repository.
- *getAtomicModelFromRep* & *getCoupledModelFromRep*: search the repository for a model matching the passed in model name.

The *databaseServices* Package also contains a class for each business object. Each class provides database services for the related business object. The majority of the methods in these classes are similar to the methods of the DAO classes. In addition some of the DatabaseService classes contain methods that are used to perform more complicated functions such as more advanced searches of the database:

- *findModels* is used by the Presentation layer to find the models, both coupled and atomic, that match certain search criteria (name, description, title, etc.). In addition it takes in an indicator to whether an OR or an AND operation should be used between these search criteria.
- *refineModels* is used to do a "refine search" operation on a set of Model search results.
- *findExperiments* is used to find the Experiments that match certain search criteria.
- *refineExperiments* is used to do a "refine search" operation on a set of Experiment search results.
- *createExpFrameWithInputsOutputs* is used when storing the EF's input and output port information. This

method creates a new EF object and assigns to it the values for the inputs and outputs that were entered by the user. It needs to check the database for existing input/output elements. If an element of an input or output is found in the database, then that element is itself used in the current EF object without having to create a new one, preventing redundancy.

5.4. Presentation Layer Java Packages

CD++ Repository is a component of the CD++ Builder, which in itself is an Eclipse plug-in. Therefore the interface to the CD++ Repository has to be integrated with the rest of the CD++ Builder Toolkit plug-in and should have a similar look and feel as the Eclipse environment. To that end the Presentation layer of the CD++ Repository is mostly built using the Eclipse Standard Widget Toolkit (SWT) package. Also, to complete the integration, the user preferences for the CD++ Repository application are integrated in the Eclipse preferences menu under the CD++ Builder Preferences section. Finally, launching CD++ Repository is done through a button in CD++ Builder toolbar in Eclipse.

5.5. Utility Classes

There are a number of utility classes that are used throughout the CD++ Repository application, collected in the *Repository.Util* package. The following is a list of the more important of these utility classes:

- *BirtUtil*: it contains a number of static methods that are used to generate the information displayed on the search results page for Experiments and Models. The class makes use of the BIRT engine and the appropriate BIRT report design files to generate HTML formatted reports that are displayed in the search results page.
- *CdxxSftpClient*: it implements a secure FTP client capable of uploading and downloading files to the FTP server specified by the IP address in the configuration.
- *FileSystemUtilities*: it class contains a number of file system related methods that are used in various places in the application.
- *MaFileParser*: it is used to parse the model definition (.ma) files for each model. By parsing the (.ma) file this class is able to build the tree of models that exist under a coupled model and to determine the input output ports for any model. It has a number of other methods that query the tree of models built by the parser.
- *ZipUtil*: it contains methods that are capable of compressing and uncompressing files using the java.util.zip utility.

6. CONCLUSION

We presented the design and implementation of a new CD++ Builder application as part of the CD++ Builder Toolkit, namely the CD++ Repository. This application is intended to make the task of re-using DEVS models easier. The difficulties that need to be overcome to enable efficient model re-use include the following:

- The availability of the re-usable components from which the modellers can easily locate the models of interest to them.
- For each model, the availability of information about the context within which a given model is valid.
- For each model, the availability of any experiments that were used to verify the validity of the given model.
- The ability to access all of this information from different geographical locations to allow teams of modellers at different locations to work together.

CD++ Repository stores all information on a server accessible over the internet from any location with internet access. It efficiently stores CD++ DEVS model information in a hierarchical structure that parallels the natural structure of DEVS models. It also stores an EF for each stored model. The EF includes information about the context of use for the stored model, as well as any number of experiments that were used to test different aspects of that model.

References

[1] Zeigler, B.; H. Praehofer; T.G. Kim. 2000. *Theory of Modeling and Simulation*, 2nd Edition. Academic Press, San Diego, CA.

[2] Chidisiuc K. and G. Wainer. 2008. “CD++Modeler: A Graphical Toolkit to Develop DEVS Models.” Poster Paper. In Proc. of SpringSim’08. Ottawa, ON. 2008.

[3] Wainer G.; N. Giambiasi. 2001. “Timed Cell-DEVS: Modeling and Simulation of Cell Spaces.” In Discrete Event Modeling & Simulation: Enabling Future Technologies. Springer-Verlag.

[4] Zeigler, B. 1984. *Multifaceted Modeling and Discrete Event Simulation*. Academic Press, London.

[5] Barros F. J.; A. Lehmann; P. Liggesmeyer; A. Verbraeck; B. P. Zeigler. 2006. “04041 Abstracts Collection -- Component-Based Modeling and Simulation.” In Proceedings of Dagstuhl Seminar 04041 Component-Based Modeling and Simulation.

[6] Traore M.K., and A. Muzy. 2006. “Capturing the Dual Relationship Between Simulation Models and Their Context.” Simulation Modelling Practice and Theory, Vol. 14, No.2, (February): 126-142.

[7] Bauer C.; G. King. 2005. *Hibernate In Action*. Manning Publications Co., Greenwich, CT.

[8] Weathersby J.; D. French; T. Bondur; J. Tatchell; I. Chatalbasheva. 2006. *Integrating and Extending BIRT*. Addison-Wesley.

[9] Wainer G., 2002, "CD++: A Toolkit to Define Discrete-Event Models". Software - Practice and Experience, Vol. 32, No.13, (November): 1261-1306.