A Common Information Model (CIM) Toolkit Framework Implemented in Java

Alan W. McMorran, *Student Member, IEEE*, Graham W. Ault, *Member, IEEE*, Ciaran Morgan, Ian M. Elders, and James R. McDonald, *Senior Member, IEEE*

***Abstract*—The common information model (CIM) is an object-oriented representation of a power system and is used primarily as a data exchange format for power system operational control systems. CIM has the potential to be used as much more than an intermediary exchange language. This paper explores the possible use of CIM as the core of a power systems analytical toolkit for storing, processing, extracting, and exchanging data directly from CIM objects. To this end, this paper discusses solutions to some of the challenges in storing and processing large power system network models as native Java objects without sacrificing reliability and robustness. This paper highlights the advantages provided by such a system when dealing with extensions to the CIM standard and overcoming the problems posed with simultaneously maintaining backward compatibility without sacrificing a higher level of detail. This paper also addresses the issue of data processing performance in contrast to other approaches.**

***Index Terms*—Common information model (CIM), extensible markup language (XML), Java, object-oriented data modeling.**

I. Introduction

WHILE data in the common information model (CIM) format encoded in the extensible markup language (XML) is set to become increasingly popular for storing and transferring power system data, CIM itself provides more than just a data storage and transfer format. The CIM specifies classes and attributes for each component of a power network, as well as defining the relationships between classes, thus making it an ideal template for both the internal storage and external transfer of data in the next generation of power system analysis tools.

This paper describes a demonstration object-oriented power network data storage system based on the CIM implemented in the Java language designed to allow simultaneous remote access from multiple users and the benefits of using such a system as an alternative to a more traditional database storage system. Section II provides a background description of the CIM class structure and how these relate to Java classes. Section III describes the structure of the proposed power system toolkit and then, in Section IV, how a CIM-based Java-coded power systems toolkit can operate to import, export, and store CIM data as well as the benefits of this approach for implementing a remote multiaccess system. Section V provides an example of how CIM can be extended within the toolkit for storing additional data, and a case study detailing how the toolkit can be used to extract data for a load flow analysis application is presented in Section VI.

Manuscript received October 6, 2004. Paper no. TPWRS-00538-2004.

A. W. McMorran, G. W. Ault, I. M. Elders, and J. R. McDonald are with the University of Strathclyde, Glasgow G1 1XW, U.K. (e-mail: a.mcmorran@eee.strath.ac.uk).

C. Morgan is with the National Grid Transco, NGT House, Warwick CV34 6DA, U.K. (e-mail: ciaran.morgan@ngtuk.com).

Digital Object Identifier 10.1109/TPWRS.2005.857846

II. Background

A. CIM

The Electric Power Research Institute's (EPRI) CIM [1] was developed as a platform independent model for describing power systems and, in November 2003, was adopted as an IEC standard [2]. CIM defines each component of a power network as a separate class but also determines how the classes relate to each other. The model itself does not describe functionality, only the associations for a class, the data it contains, and the format of the data.

This approach of creating a generic model of power network components allows the model to be translated into classes for exploration and manipulation within an object-oriented programming language application. Languages such as Java, C++, and C# are object oriented, and the CIM structure can be used to create corresponding classes in any of these or other object-oriented languages. Each CIM class becomes a corresponding class in the target language, with the corresponding attributes and inheritances.

B. Power System Analysis Software Design Methodologies

Much of the power system simulation software currently in use within large utility and consultancy organizations is based on procedural programming languages, many of which date back to the late 1970s and early 1980s, when processing power and memory capacities were a tiny fraction of what is available today. Since then, programming languages have evolved, moving on from procedural, functional programming to an object-oriented design. This technique involves creating programs as instances of modules or classes, so that a program is split into a number of small, self-contained systems that are adaptable and reusable.

As mentioned, CIM is ideal as the basic framework of such object-oriented software, since it defines all the basic components of the power network as objects that can be quickly converted to classes in an appropriate object-oriented programming language application. All the different components of a network can be instantiated simultaneously, creating a dynamic computer representation of the network.

Since components within a power network do not operate independently, a data model must reflect the relationships between classes. With object-oriented software, there are three

0885-8950/$20.00 © 2005 IEEE

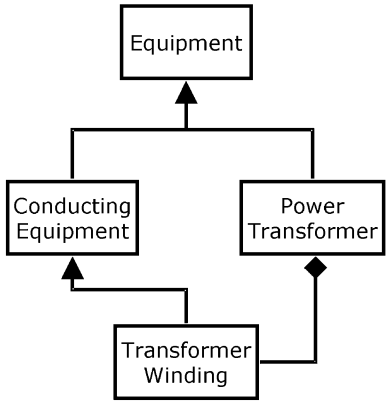

Fig. 1. Sample UML representation of a sample CIM class structure.

main types of relationships between classes: inheritance, aggregation, and association.

With inheritance, a class takes on all the properties of its parent, as well as adding values and methods of its own. For example, as shown in Fig. 1, the power transformer winding class inherits all the attributes of the conducting equipment class, which itself inherits all the attributes of the equipment class.

Aggregation denotes a relationship where one class is constructed using instances of another class that exists only as part of the parent; for example, the power transformer class is made from two or more transformer winding classes. The windings exist only as part of the power transformer and, in the case of a power system model, would not exist as stand-alone objects.

Association indicates that two independent classes are linked, allowing the bidirectional flow of data, but the classes can exist without each other. For example, the steam turbine object will be associated with a steam supply object, since they have a relationship (supply of steam from steam supply to steam turbine) but can exist independently of each other.

With a CIM implementation of a power system, objects can communicate directly with each other, reducing the requirement for a single, centralized program or procedure to perform all the processing of data. Multithreaded software, where more than one process can run concurrently, based on an object-oriented CIM network, would allow a distributed simulation system that integrates the processing and storage of data.

The use of the CIM for the basis of the underlying architecture instead of a custom-designed solution as has been used in other object-oriented power system applications [3] to allow the framework to cope with extensions and modifications to the CIM standard without requiring a complete redesign of the software. The CIM defines the relationships between classes and by defining rules for creating functions within each class for setting and getting the attributes or associations. The API for the software follows a pattern based on the unified modeling language (UML) standard, prefixing an attribute name of $0..1$ association with get and set and prefixing add and remove to $0..n$ associations. This is a common practice for object-oriented software development and allows for simple creation of Java classes from a UML design and, as such, allows the software to integrate extensions to the CIM standard with little modification. A custom-designed class hierarchy is unlikely to integrate extra CIM classes without significant redesign of the class structure.

Many existing power systems analysis software packages and energy management systems can import and export data in the CIM format, encapsulated in XML. However, they convert this data into or from their own internal data structure, and as such, the ability to cope with extended formats or modifications to the standard is obviously limited by how the CIM maps to their own data structure.

III. Power Systems Toolkit Design

The CIM objects form the core data storage system for the power systems toolkit proposed in this paper. Access to the data is via the core toolkit module's application programming interface (API), which maintains the integrity of the data. Additional modules can be attached to this main module, allowing the import, export, and modification of the data.

The primary motivation for the design of this system is to allow it to operate as a remote application, providing multiple users access to the same data and tools concurrently. This system creates problems concerning the synchronization of data between multiple, concurrent user sessions. A remote system of this type must prevent multiple sessions from creating multiple instances of the original data if any changes made are to be integrated back into the original data during runtime. These issues will be addressed further in Section IV.

Fig. 2 shows the structure of the toolkit: the import module, the core toolkit and its associated serialization module and object storage system, and three additional modules that utilize the API. The export modules (PSS/E in our case but could be any other power systems analysis application) use the API to read the core data and then process it into the required output format. The topological processor uses the existing CIM objects to create associated topological node objects, which are inserted into the core data storage system, via the API.

The import module has access to the CIM class definitions and must be able to determine which classes are required and how many instances of each class will be required to successfully parse all the data from the source CIM XML file. The module will then work its way through the imported file, creating instances of one or more CIM classes for each component of the network described in the source file, until the whole network has been instantiated as interconnected CIM objects.

This CIM model is then passed to the core toolkit, via the API, which integrates these objects into its core object storage system. The additional modules can then access the data through the core module's API. This way, the data can be interrogated, modified, or exported, while the core module maintains the integrity and tracks modifications to the data.

Accessing the data through the core model will allow the implementation of the serialization-based journaling system described in Section IV-E, since the data itself are not made available natively to the attached modules but is accessed through the core API. This records, and serializes, all commands performed that modify the core data, as well as verifying the integrity of the attached modules, to prevent unauthorized access to the sensitive network data.

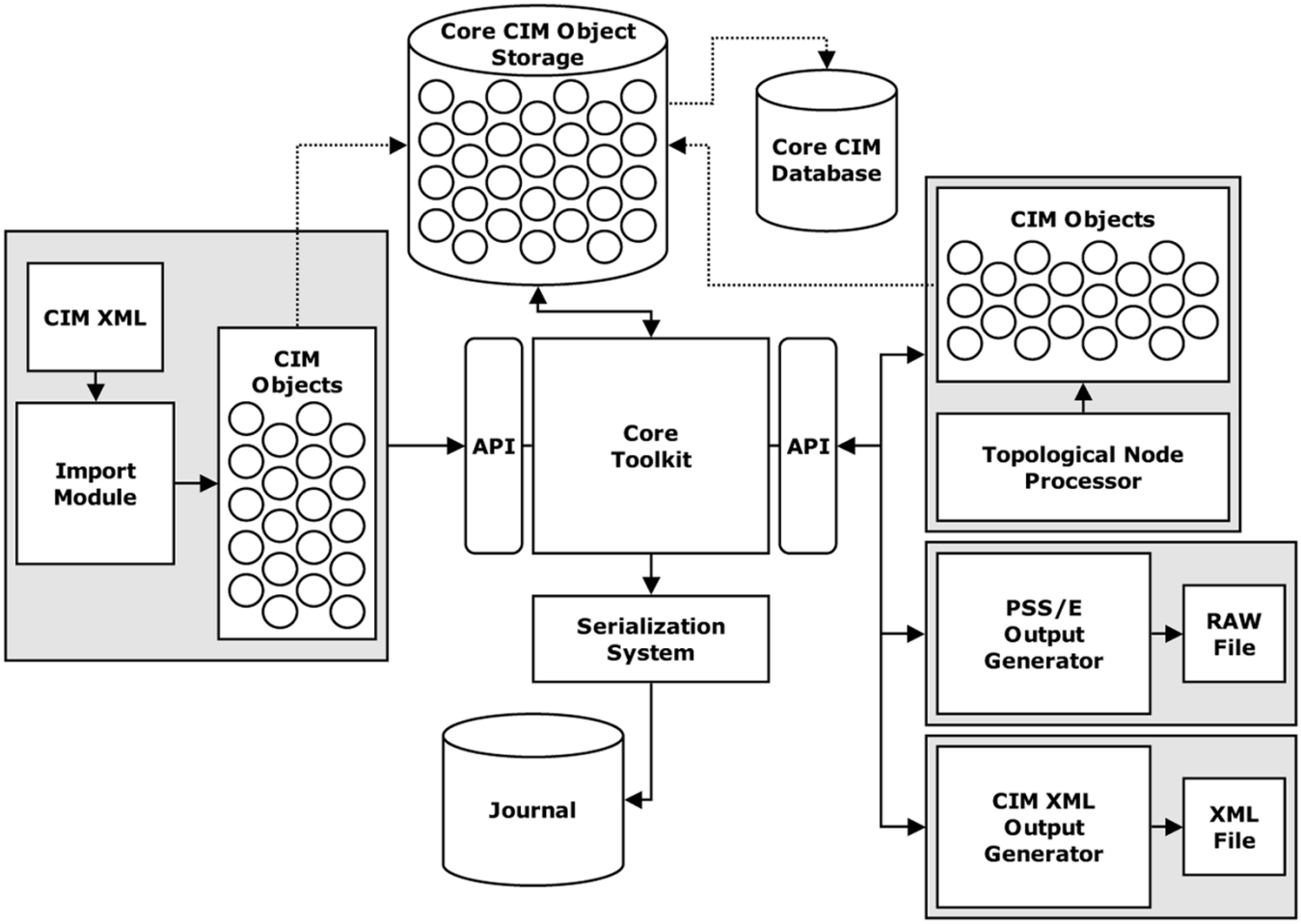

Fig. 2. Structure of core toolkit showing interaction with external components via API.

IV. Features of Implementing a CIM-Based Object-Oriented Power System Toolkit

Several challenges were overcome in creating a CIM-based power systems toolkit in Java. These included creating a Java implementation of CIM, choosing a system of data storage for the toolkit that would be both fast and flexible and allow concurrent access from multiple sources, creating a simple method of importing data from a CIM XML file into the toolkit itself, and ensuring that the software system was reliable.

A. Implementation of CIM Classes in Java

An implementation of CIM in the Java programming language was undertaken to demonstrate the expected advantages of using objects to store network data. Since Java is a fully object-oriented language, the UML [4] version of CIM already available [1] could be used to automatically create the base Java classes. There are tools available, both freestanding and within existing commercial design packages, to translate the language-independent UML specification into a number of programming languages and create the appropriate base functions for inserting and retrieving values from each class.

The benefit to using the CIM for the software's internal architecture rather than a custom solution is that with major power systems software vendors such as ABB and Siemens implementing CIM import and export functionality into their software, storing the data internally in the CIM format removes one extra level of translation when exchanging data between applications. The ability to cope with extensions to the CIM is also easier when the internal data storage architecture mirrors that of the standard. Any additions and modifications to the standard requires only a corresponding change to the internal data storage architecture, removing the problems of mapping a proprietary internal architecture to changes in the CIM.

When converting the CIM UML classes into Java code, functions are automatically created for adding, modifying, and retrieving data in the format of get[Attribute]() and set[Attribute](), where [Attribute] is replaced with the name of the attribute, or get[Associations](), add[Association](), and remove[Association](), where similarly [Association] is replaced with the name of the association. This allows the classes to follow the standard but can also be easily extended to create custom classes with additional custom functionality. These functions allow a single function call to perform multiple data modifications on the object and any other associated objects. This combines the data storage and manipulation into one single entity, which, while more complicated than a file or database system, provides a more comprehensive API and

enables the actual software applications using the data to be simplified.

Java's performance, and thus suitability, for computationally intensive applications can be measured using the Math, Statistics, and Computational Science Division of the National Institute of Standards and Technology's SciMark2 benchmark suite [6]. This benchmark is used to measure the performance of computer systems using a series of computational kernels: a fast Fourier transform, Jacobi successive over-relaxation, Monte Carlo integration, sparse matrix multiply, and dense LU matrix factorization. The benchmark code is available in both C and Java and returns values in million floating-point operations (MFlops) per second for each kernel and a composite score.

The two matrix-based benchmarks offer the most relevant comparison for this application, since the extensive use of arrays in each kernel allows the access times of the two systems for storing arrays in each language: object references in Java and pointers in C, to be compared as well as the numerical performance. Benchmarks run recently on a Pentium 4 system using both the Java and C versions found that while the C code was marginally faster overall, producing a composite score of 384 MFlops compared with 361 MFlops for the Java 1.4.2 version, this is a drop of only 6%. The full benchmark results can be found in [7].

The advantages of using Java are its cross-platform compatibility, the ability to easily use the same framework for graphical, command-line or server-based applications, the inbuilt libraries for constructing distributed applications, the intrinsic security features of Java, and the integrated multithreading capabilities, all of which outweigh what is now, with modern versions of Java, only a small performance disadvantage over natively compiled C or C++ code.

B. Advantages of Storing a Power System Model as Objects

The traditional approach to storing large quantities of data for concurrent access from multiple sources is to use a database system for storing, searching, and retrieving the data. Such databases required the use of the database interface, which, while powerful for complex searching, has significant disadvantages when performing traversals of the power system network layout, like those required for converting node-breaker data in bus-branch format, at a higher level of abstraction.

Using native objects for persistent storage offers several benefits as follows.

- Storing the data as native objects allows far greater flexibility for access and manipulation of data.
- The interface is written in the native language and is fully customizable, so data manipulation and access can be fully integrated into the data storage medium.
- A standard data format can be maintained while providing a powerful, adaptable interface for accessing the data.

The use of packages and inheritance allows for the core classes to be extended, but compatibility with previous versions to be maintained by using separate packages for each version of the CIM and the use of inheritance between packages allows backward compatibility to be maintained where required.

One of the main functions of the toolkit is to automatically create topological node objects, analogous to a bus in bus-branch format, from the more detailed node-breaker format of CIM. This process has been described in a previous publication [5], along with the implemented algorithm. For comparison, the algorithm was implemented using an object-storage system for the CIM data and then using a MySQL database to store identical data.

This comparison is due to the three choices available for a multiaccess system: Using a database to store the data and allow it to be accessed concurrently by each session as data is required, use a persistent object-storage system for common data storage, and use a database for data storage then instantiate an independent set of objects based on the database data for each user session. The latter option creates the problem of synchronizing data between multiple sessions but, for the purposes of benchmarking, operates on the same principle as the object-storage option.

With the object-storage option, the references to any connected objects are contained within the object as a direct reference. With a database, the field that refers to another entry in the database uses a foreign key to locate the entry for the other object and extract the appropriate information.

Assuming the database is fully indexed, containing data for n objects with each object having a single row and unique numeric identifier in the primary table, then a single step in a network traversal would occur in $\mathrm{O}(\log n)$. This is because a standard binary search of an ordered index would take place on average in $\log n$, for a fully ordered index of n items. If the traversal takes m steps, then the database will take $\mathrm{O}(m \log n)$ to complete the traversal.

For a persistent object-storage system, a single step of the traversal will take $\mathrm{O}(1)$, since objects contain direct references to other associated objects, and no searching is required. A full network traversal therefore takes place in m steps, or $\mathrm{O}(m)$.

It can therefore be concluded that for a network of size n, the time taken for a complete network traversal with an object-persistent storage system will see a linear growth as the number of steps in the traversal m increases but is independent of the network size n. Using a database system for data storage and retrieval, however, shows a growth that is linearithmic, affected by both the size of the network and the average number of steps for a traversal.

C. Memory Storage Requirements for an Object-Based System

An object-prevalent system stores all the data in memory. To compare the memory requirements when instantiating a power systems model as CIM objects in Java, the toolkit instantiated an increasing number of power system models in memory and used Java's Runtime $\cdot$ totalMemory() to measure the memory used by the Java virtual machine.

The test model used was the Siemens 100 bus model from the CIM interoperability tests, a network of 6976 objects representing a network of 100 buses, 55 generating units, 77 loads, and 137 lines. By making the toolkit instantiate multiple instances of this model, the amount of memory used to store networks of increasing size can be measured.

On a computer with 1 GB of physical memory, the memory usage follows a linear growth pattern; however, due to operating system overheads, beyond 800 MB of used memory, the system's responsiveness decreases as paging to virtual memory takes place.

The toolkit was tested on a 1.5-GHZ G4 Unix system with 100 instances of the Siemens model instantiated simultaneously, creating close to 700 000 objects, representing a network of 10 000 buses using a reported 1 GB of memory. With the presence of additional physical memory, there is no evidence that the system would not linearly scale upwards until the 32-bit memory address limit is reached at 4 GB. Beyond this, Java 1.5 supports 64-bit memory addressing, and with modern 64-bit workstations and servers available at prices less than $3000, the system has the potential to scale up to tens or hundreds of millions of objects. Beyond this limit, the system could be split across multiple computers, operating as an interconnected cluster. This, however, is beyond the scope of this paper.

An alternative is to implement persistent CIM Java objects using a database with Java database connectivity (JDBC), which stores each object in a serialized form within a database. The problems with such a system have been discussed previously [8], where it was noted that the complexity of mapping Java objects into tables and back increases dramatically when multiple levels of inheritance, as contained in CIM, are included.

The other system used in the aforementioned paper [8] was an object database management group (ODMG) Java binding, which has since been superseded by Java data objects (JDOs). This is a system for storing Java objects in a database with transparent object mapping to database tables. While this approach does offer advantages over JDBC since it is simpler to implement, it still requires a database system to function, which removes much of the speed advantage of an object prevalent storage system since each object is loaded from the database.

A database storage system does offer advantages when performing complex searches across multiple object types. For example, locating all pieces of equipment within a specific voltage level would require a single database query in the database implemented previously, but to implement the same search in an object-storage system would require functionality to search through every instance of each equipment type to find any matches.

The advantages provided by the superior search capabilities of a well-designed database make the combination of an object-store with an associated read-only database the most attractive option. Making the database read only and making any changes to data within the object store be automatically mirrored on the database by each object removes the problem of synchronizing data between the two. This system provides the advantages of complex database searches while maintaining the benefits in speed for complex data transformation provided by an object-storage system.

D. Importing CIM XML Power System Data Into Java Objects

As has been mentioned previously, CIM provides a framework for creating an object-based representation of a power system. Rather than interpreting CIM XML [9] data directly, importing this data into a series of CIM objects allows greater flexibility in accessing and manipulating the data. Of course, the construction of an export module for straight translation into other proprietary power system modeling data formats is also possible.

The model is instantiated using a generic import module that can cope with any XML components within the document that validate against the defined schemas and for which a corresponding class file exists. Rather than using a large compare statement to locate the corresponding class, the name of the XML components can be used to create a class file of the same name, using Java's reflection functionality. This allows the available functions of a class to be found dynamically by the program when running. As previously described in Section IV-A, if a class is constructed so that for a variable X, the corresponding set and retrieval methods are setX and getX, then it is possible to propagate the object based on the data retrieved from the XML component.

This way, small pieces of code can be reused for importing any CIM XML node, since the import module is generic and nonclass specific. As each XML component is read, the name of it is used to create an object whose class has an identical name to that of the XML component. Now the attributes in the XML component can be added into the object by using the reflection API to locate a set method that matches the name of the attribute.

For example, the code below shows an excerpt from a CIM XML file containing the g, r, x, ratedKVA, and ratedMVA values for a transformer winding, TW_1A.

```
⟨cim:TransformerWinding rdf: ID="TW_1A"⟩
  ⟨cim:TransformerWinding.g⟩0.04
  ⟨/cim:TransformerWinding.g⟩
  ⟨cim:TransformerWinding.r⟩0.07
  ⟨/cim:TransformerWinding.r⟩
  ⟨cim:TransformerWinding.x⟩0.47
  ⟨/cim:TransformerWinding.x⟩
  ⟨cim:TransformerWinding.ratedKV⟩400
  ⟨/cim:TransformerWinding.ratedKV⟩
  ⟨cim:TransformerWinding.ratedMVA⟩164
  ⟨/cim:TransformerWinding.ratedMVA⟩
⟨/cim:TransformerWinding⟩
```

When these data are imported into the toolkit, the import module selects the corresponding Java class, TransformerWinding by using the name of the XML node, cim:TransformerWinding, locating the Package in the CIM that contains the TransformerWinding class then using Java's reflection functionality to create a new instance of that class. Given that the Java class structure and XML schema are created from the same UML model, an XML file that validates to the schema during the initial stage of importation will in turn map to one of the class files, preventing the software from trying to import invalid XML data.

The Java class contains attributes that correspond to the values in the XML nodes: variables named g, r, x, ratedKVA, and ratedMVA and also contains the functions setG, getG, setR, getR, setX, getX, setRatedKVA, getRatedKVA, setRatedMVA, and getRatedMVA that are used to set or retrieve the corresponding variable's value.

The data will initially be in a string format since an XML file is plain text, but this can be converted to the appropriate data type within the class itself during the set method, keeping the import module simple and generic. For example, the string "34.2" can be converted to the floating-point value 34.2 by the set method, since the variable in the class will be a floating-point value type rather than a string.

For object associations (where the attribute of the XML node refers to another node), it is necessary to initially store these associations as the unique string identifier for the other component. As each node is read in, an index is updated with the identifier of an object and a reference to the object itself. Then, once the file has been fully imported and all the objects instantiated, the references in each object can be converted from a text identifier to an object association using the previously created index.

This import system results in the saved network model being instantiated as CIM Java objects, and thus, any export or processing module can access and modify the data through each object's API. The associations and interconnections between the objects simplify the process of modifying multiple interconnected objects and provide a powerful API, capable of much more than simply setting or retrieving the data. A single command could result in changes to data rippling through all the associated objects automatically as functions in one object can call additional functions to modify data accordingly in associated objects.

E. Use of Serialization to Track Model/Data Changes for Security

One of the major disadvantages of storing the model in memory is that in the event of a system crash, the entire model is lost, since memory is, in the majority of cases, volatile, and the data will not be recoverable. This drawback, however, can be overcome, without significantly reducing the advantage provided by object prevalent data storage.

Java, and other common object oriented languages (C++, C#, CORBA), have support for serialization (saving the state of an object into a file). It is beyond the scope of this paper to explain the intricacies of serialization; however, the basic process involves converting the current state of an object as an encoded stream of bytes, which can then either be saved to a file or transmitted. This byte stream contains all the data required to then reconstruct the original object. Serialization can also be applied to the commands executed on the model, so a record of these commands, combined with a regular serialization and storing of the model's state at a given point in time, allows for the full restoration of the model's current state following a system crash.

For saving the state of the model during a controlled shutdown of the system, however, the model's state can be exported as a valid CIM XML file. This allows any upgraded version of the system containing a newer version of the CIM class hierarchy, with additional or modified classes to recreate the full network from the CIM XML file.

An object-prevalent architecture for Java, using this serialization system Prevayler [10] is available as an open source Java layer under the General Public License. This provides the framework for implementing an object-prevalent system. Some more information on the object-prevalence concept and examples of its implementation with Prevayler can be found in [11].

V. Extending CIM

It had been noted elsewhere [12] that CIM, while comprehensive in many areas, lacks the detail required for specific areas of power system engineering. The object-based design of CIM allows for enhancements and modifications to the current standard, but it can be easily recognized that *ad-hoc* modifications to a standard are undesirable in the majority of cases. Open standards, such as CIM, are adopted to aid the process of exchanging data, and so, data in the standard format and structure (or a future revision) immediately becomes incompatible if the standards used are subsequently modified.

Using an object-prevalent data storage system, however, can provide a compromise, allowing enhancements and extensions to CIM data, but still maintain the capability to output CIM 1.0-compliant data. Extensions to the CIM are defined in UML, and by autogenerating the appropriate class files from this UML file, these new classes can be easily integrated into the existing data storage architecture without requiring modifications to the existing importation module.

The most basic way of adding additional data is to add extra attributes to each object in addition to the standard CIM attributes. This allows for more data to be stored in each object while maintaining the ability to export the standard data without introducing incompatibilities.

To extend the level of detail that can be stored in object format, it may be advantageous to have child classes for existing classes; for example, a line object is currently made up of one or more alternating or direct current line segments in CIM 1.0 representation. In some cases, having the resistance and reactance specified as absolute values for each line segment is not sufficient for some users of CIM, and it may be better for this value to be calculated based on the following:

- length of each line segment;
- type of conductor and bundles;
- type of tower;
- number of towers for each segment;
- distance between towers;
- positioning of phases on each tower;
- number of circuits using each tower.

Some of this data is not currently included in standard CIM; however, additional child classes and attributes could be added to the line and line segment classes to allow this data to be stored. Using objects, when the value of, for example, the resistance is requested from an ac line segment object, with standard CIM, the value is an absolute value stored as a variable within the object. Using the enhanced data objects, however, the value returned is itself calculated by a function within the object based on the values of both the object's internal values and the child objects associated with it. This way, CIM compliance is maintained, since the resistance value is still obtainable if desired, but the greater level of detail can be maintained and is available by direct interrogation of the child objects. The ability to include multiple schemas in the same XML file also allows for the standard and enhanced data to be included in a single output file, without breaking compatibility.

Fig. 3 shows a section of the line model in CIM with additional classes as an example of how information on towers can

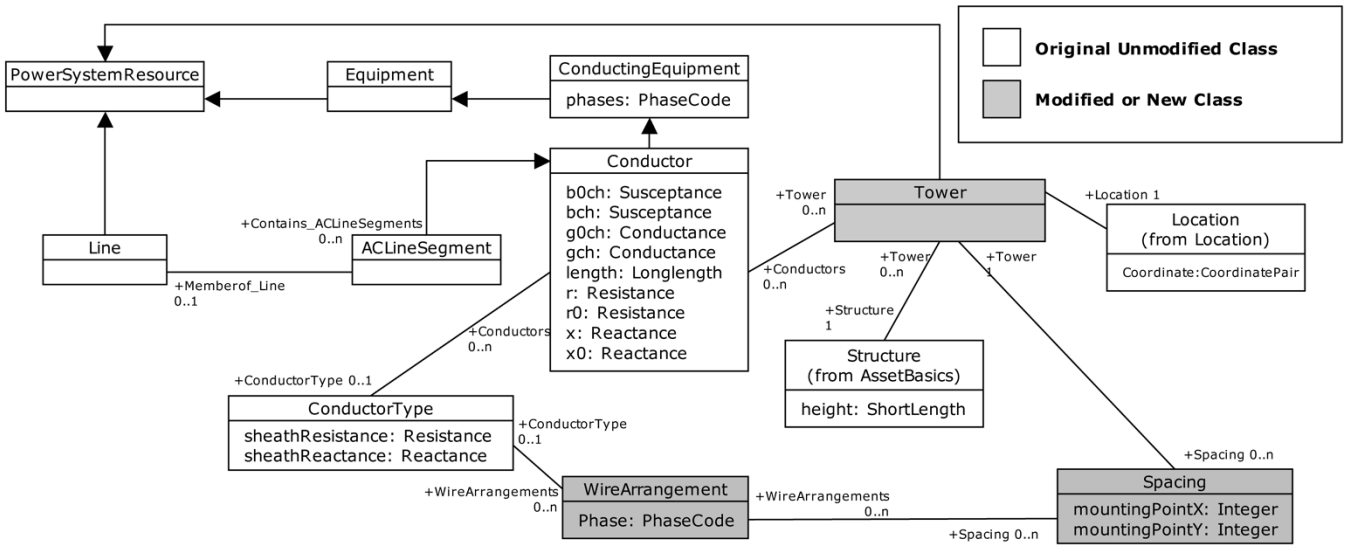

Fig. 3. Example extensions to CIM for tower connections.

be added into the data model. This allows the identification of points where two or more circuits share a tower, which can affect the electrical characteristics of the individual phases of a circuit. A line will span several towers, at varying distances from each other, and the span between each tower will be defined as an instance of the conductor class. Each tower on the line is recorded with a single instance of the tower class, itself associated with all instances of conductor from every circuit that uses the tower. A resistance value for the conductor would be calculated using the attributes of its associated classes (conductor type, wire arrangement, wire type, and any instances of the tower class) as well as the positioning of the phases from any surrounding conductors. This would allow a getResistance function in a conductor object to calculate the resistance each time the function is called. This way, any changes to the attributes of any child classes or the nearby conductors would always be reflected in the value of resistance returned from the conductor object.

Using a generic import system as detailed previously allows for additional classes to be added by simply including their class descriptors in the configuration settings for the toolkit. This way, the import modules and toolkit itself do not need to be recompiled or rewritten to cope with additional classes. The network model can contain additional information in this extended format, and by ensuring that the version information for each class is correctly assigned, the model can be exported with or without the additional data included.

By structuring the extensions in such a way that, wherever possible, the extra data and classes do not break the existing standard's class associations, and then, these additional classes will enhance the existing data without necessarily disrupting backward compatibility.

VI. Case Study: Power Systems Object-Oriented Toolkit PSS/E Export Function

Since CIM is still establishing itself as a standard for power system network data (and is primarily intended for exchange of data for power system operations), there is a need to convert CIM XML data into proprietary formats for use with legacy, offline power system analysis applications. For many applications, the conversion can be done with a simple extensible stylesheet transform (XSLT), but for more complex data formats, such as that of PTIs PSS/E, a more sophisticated conversion is required.

Additional modules for the toolkit have been created: one to generate topological nodes and identify islands within the network and the other to export the data as a PSS/E RAW data file. The first module complements the second, since the topological nodes are created using an algorithm developed by the authors, which amalgamates nonprimary network components into a topological node, which is analogous to a bus in bus-branch formats such as PSS/E. This work has been detailed in a previous publication [5].

The PSS/E export module can access the new topological nodes data and use them to create bus identifiers required for creation of the PSS/E compatible output, as well as identifying all the generators and loads connected to each bus. The PSS/E export module then extracts the appropriate information from the model and uses it to construct objects of its own to represent the bus, branch, transformer adjustment, generator, and load data, all of whose values are stored as associations to existing data within the CIM objects. These PSS/E export objects can then export their data as a string of numbers for insertion into the output file. Any changes to the underlying CIM model results in an automatic corresponding change to the PSS/E export objects. This is only one example (already demonstrated successfully) of export from an object prevalent CIM model to an external application software package.

VII. Conclusion

The combination of several new technologies, notably the CIM and object-prevalent data storage, along with established techniques for object-oriented programming, has created the foundations for an international standard-based, highly extendable, and scalable system for storing and manipulating power system data. The implementation of the framework in Java,

which is available on a wide variety of computing platforms, allows for significant platform independence for the system.

The use of the open CIM standard for the internal software architecture is itself a novel concept, since the standard is currently used as an intermediary exchange format between applications that have their own internal storage architecture.

Combining the data storage and manipulation into a single memory resident program for long term storage, while unconventional, provides significant speed advantages over dealing with a native file or database when performing complex interrogations of the power system. It allows for the development of a powerful and significantly more extendable API than that of a traditional enterprise-level database. Using Java allows the software to be deployed in a number of ways, including, with little change, a web servlet, allowing remote execution of the software. This aids the development of web services and other remote or distributed systems for power system simulation and analysis based around the toolkit. Web services have been highlighted previously as being of possible benefit for remote access to power system simulation software [13], and the use of Java provides this simple path for deploying the toolkit as a web service.

The system provides functionality for the data to be easily exported in XML format, allowing the exchange of data between applications and companies in an open nonproprietary format, which is one of the major reasons for the development of the CIM in the first place.

The ease with which modules to modify and export the data can be constructed highlights the flexibility of the design and the scope for extending and utilizing the core framework in the future.

REFERENCES

[1] EPRI, Common Information Model (CIM): CIM 10 Version, Palo Alto, CA, 2001. 1 001 976.

[2] "Energy Management System Application Program Interface (EMS-API)—Part 301: Common Information Model (CIM) Base," IEC, version 1.0, Nov. 2003.

[3] M. P. Selvan and K. S. Swarup, "Object methodology," *IEEE Power Energy Mag.*, vol. 3, no. 1, pp. 18–29, Jan.–Feb. 2005.

[4] Unified Modeling Language Specification, OMG. (2001). [Online]. Available: http://cgi.omg.org/docs/formal/01-09-67.pdf.

[5] A. W. McMorran, G. W. Ault, I. M. Elders, C. E. T. Foote, G. M. Burt, and J. R. McDonald, "Translating CIM XML power system data to a proprietary format for system simulation," *IEEE Trans. Power Syst.*, vol. 19, no. 1, pp. 229–235, Feb. 2004.

[6] "SciMark 2.0", Nat. Inst. Standards Technol.. Maths, Statistics, and Computational Science, Gaithersburg, MD. [Online]. Available: http://math.nist.gov/scimark2/.

[7] Performance Comparison of Java/.NET Runtimes (SciMark 2.0 in Java, C# and C), Kazuyuki Shudo. [Online]. Available: http://www.shudo.net/jit/perf/#scimark2.

[8] D. Barry and T. Stanienda, "Solving the Java object storage problem," *IEEE Comput.*, vol. 31, no. 11, pp. 33–40, Nov. 1998.

[9] A. deVos, S. E. Widergren, and J. Zhu, "XML for CIM model exchange," in *Proc. Power Industry Computer Applications Conf.*, Sydney, Australia, 2001.

[10] Prevalyer [Online]. Available: http://www.prevayler.org/.

[11] C. E. Villela, "An Introduction to Object Prevlance,", http://www-106.ibm.com/developerworks/library/wa-objprev/..

[12] X. Wang, N. N. Schulz, and S. Neumann, "CIM extensions to electrical distribution and CIM XML for the IEEE radial test feeders," *IEEE Trans. Power Syst.*, vol. 18, no. 3, pp. 1021–1028, Aug. 2003.

[13] A. W. McMorran, G. W. Ault, G. M. Burt, and J. R. McDonald, "Web services platform for power system development planning," in *Proc. University Power Engineering Conf.*, Thessaloniki, Greece, Sep. 2003.

Alan W. McMorran (S'02) received the B.Eng. degree in computer and electronic systems from the University of Strathclyde, Glasgow, U.K, in 2002. He is currently working toward the Ph.D. degree at the University of Strathclyde.

Currently, he is a Research Student with the Institute for Energy and the Environment, University of Strathclyde, where he has been since 2001, working in the areas of model and data format exchange technologies. His research interests include the use of open standards and Internet technologies for power system planning and modeling.

Graham W. Ault (M'00) received the electrical and mechanical engineering degree (Hons.) and the Ph.D. degree from the University of Strathclyde, Glasgow, U.K., in 1993 and 2000, respectively. His thesis was on the impact of small-scale generation on electricity networks.

His research interests include power system modeling and analysis, distributed generation, asset management, and power system planning and development.

Ciaran Morgan received the First Class B.Sc. (Hons.) degree in electrical and electronic engineering from Cardiff University, Cardiff, U.K., in 1989.

He is currently working within the Network Design Department, National Gird Transco, Warwick, U.K. His responsibilities include the exchange and management of transmission system users data and transmission system analysis data files for the suite of analysis applications use by network design for long-term planning purposes. He is also involved in the specification of the next-generation analysis/data facilities for transmission analysis purposes.

Ian M. Elders received the B.Eng. and Ph.D. degrees from the University of Strathclyde, Glasgow, U.K., in 1994 and 2002, respectively.

He has undertaken research in collaboration with South Western Electricity, Exeter, U.K.; National Grid, Coventry, U.K.; and GE Harris Energy Control Systems, Livingston, U.K. His research interests include intelligent systems applications in power system control and management and application of Internet technologies in power system operations.

James R. McDonald (M'90–SM'01) was born in Glasgow, U.K., in 1957. He received the B.Sc., M.Sc., and Ph.D. degrees from Strathclyde University, Glasgow, U.K., in 1978, 1984, and 1989, respectively.

Currently, he is the Rolls-Royce Professor of power engineering in the Center for Electrical Power Engineering, Strathclyde University. He was appointed Lecturer within the Department of Electronic and Electrical Engineering, Strathclyde University in 1984. He was also with the South of Scotland Electricity Board, Glasgow, U.K., from 1978 to 1984. His experience was primarily associated with power system protection and measurement. His research activities include power system protection and measurement, artificial intelligence applications in power engineering, and energy management. He has published many technical papers and is the coauthor of two books.