THE IRMIS OBJECT MODEL AND SERVICES API*

C. Saunders, D. A. Dohan, N.D. Arnold
APS, Argonne, Illinois

ABSTRACT

The relational model developed for the Integrated Relational Model of Installed Systems (IRMIS) toolkit has been successfully used to capture the Advanced Photon Source (APS) control system software (EPICS process variables and their definitions). The relational tables are populated by a crawler script that parses each Input/Output Controller (IOC) start-up file when an IOC reboot is detected. User interaction is provided by a Java Swing application that acts as a desktop for viewing the process variable information. Mapping between the display objects and the relational tables was carried out with the Hibernate Object Relational Modeling (ORM) framework. Work is well underway at the APS to extend the relational modeling to include control system hardware. For this work, due in part to the complex user interaction required, the primary application development environment has shifted from the relational database view to the object oriented (Java) perspective. With this approach, the business logic is executed in Java rather than in SQL stored procedures. This paper describes the object model used to represent control system software, hardware, and interconnects in IRMIS. We also describe the services API used to encapsulate the required behaviors for creating and maintaining the complex data. In addition to the core schema and object model, many important concepts in IRMIS are captured by the services API.

IRMIS

IRMIS is an ambitious collaborative effort for defining and developing a relational database and associated applications to comprehensively document the large and complex EPICS-based control systems of today's accelerators. The documentation effort includes process variables, control system hardware, and interconnections. The approach could also be used to document all components of the accelerator, including mechanical, vacuum, power supplies, etc. One key aspect of IRMIS is that it is a documentation framework, not a design and development tool. We do not generate EPICS control system configurations from IRMIS, and hence do not impose any additional requirements on EPICS developers.

Typical documentation schemes utilizing revision control drawings are quite limited and not practical for managing such a complex system. Defining the relationships between thousands of elements is much more suited to a relational database implementation of documentation rather than a drawing-based method. The vision is simple; define every entity of an accelerator control system and its relationships with other entities. Once defined, numerous views of these relationships can be implemented that will answer the routine questions posed of a system during operation and maintenance. Some typical examples include:

- What process variables are associated with this component?
- What process variables were added, changed, or removed since the last run?
- Where does the other end of this cable go?
- What do all these nonfunctional components have in common?
- How many components of a particular model number are installed?
- What are all the types of components used to provide analog-to-digital conversions?
- What equipment will be affected when this circuit breaker is locked out?
- What application software will be affected if this component is removed?

IRMIS captures roughly three key groups of information as part of its documentation. One, the history of all IOC boots is maintained where each occurrence results in capturing the complete set of process variable definitions (record types and fields) and process variable instances. Two, the current configuration of control system components is captured, including both physical and logical relationships. Physical relationships between the components are represented by interconnects, either direct port-to-port, or with an intervening cable. Logical relationships between components are

* Work supported by U.S. Department of Energy, Office of Basic Energy Sciences, under Contract No. W-31-109-ENG-38.

captured by housing, control, and power hierarchies. A third schema captures the dynamic mapping between the process variables and the components [1].

Components are documented by their relationships with other components, not by an exhaustive breakdown of component subtypes and equipment-specific attributes. In essence, the component schema of IRMIS is a directed, acyclic graph of nodes (components) and edges (relationships). Only pure hierarchies are represented currently, although the schema can accommodate nonhierarchical networks of relationships.

The power and extensibility of this representation is substantial, but does require additional processing on the part of the applications. While possible, it is more difficult to create single SQL queries to derive results from the data. Application development and intermediate layers of software above the relational database take on increased importance in IRMIS.

DEVELOPMENT

Two requirements have largely forged the current implementation of IRMIS. First, the desired features for applications necessitated a thick client graphical interface. The user interaction offered by HTML and JavaScript solutions was insufficient for all but basic views of the data. Secondly, the project is to be database vendor independent as much as possible. Given the effort at any given site that goes into deploying and maintaining a relational database, it wasn't realistic to impose a particular vendor.

The initial development focus was on the relational database schema with some simple scripts and PHP pages to interact with it. As development progressed, and in particular the component schema design, it became clear we would need a complex data model and substantial business logic. A relational database is a relatively unconstrained design space in terms of representing specific data structures and semantics. While stored procedures can be used to impose additional semantics, it is largely impossible to maintain database vendor independence using stored procedures. For this reason, much of our focus is now on data modeling and business logic in the application domain. Most of this paper is subsequently focused on the application object model, and not on the details of the underlying relational schema, which provides persistence.

The project settled on the use of Java and Swing for the graphical interface, and on the Hibernate ORM for database portability and data modeling. Hibernate bridges the gap between the relational data model and the object-oriented data model. This gap is typically responsible for a significant amount of development and maintenance time. Object Relational Modeling frees the developer to focus on applications and data, not transforming between the two models.

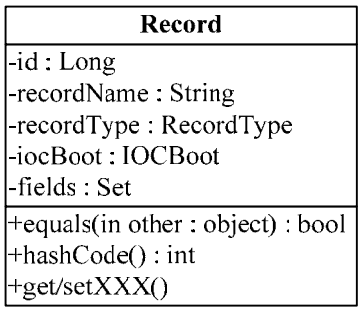
The project currently has a stable schema for process variables and a complete Java application for viewing the data (as well as Perl crawlers for populating the data) [2]. The component schema is approximately 80% complete, with Java applications already for viewing and editing much of the data. The remaining work involves extending the applications and schema to include the physical interconnections between component ports [3]. The remainder of this paper is focused on the object model to date and the services API in Java that enable these applications.

OBJECT MODEL

The IRMIS object model is simply a collection of POJOs (plain old java objects). The Hibernate ORM framework imposes no inheritance requirements on the Java classes, using instead a hidden mechanism involving introspection and proxies. The only requirements are that each class that is to be persisted contain an id property of type Long, and an equals() and hashCode() method (both of which are good practice in any environment). An XML file called a Hibernate mapping file is used to define the association between properties in the Java class and the underlying database tables and columns. This mapping capability is thorough and flexible, allowing the direct mapping of Java primitives to relational database columns, as well as mapping Java collections to associated relational tables.

Figure 1 shows a single class definition corresponding to an EPICS record, and the Hibernate mapping used to realize this object in the database. The id property of the Record class is not typically used publicly, but is instead used by Hibernate to manage the persistence lifecycle of a Record object. If the id is null, then the object is not yet persistent. If id is non-null, it typically reflects the value of

the corresponding id column of the mapped database table. The remaining properties are business data. The recordName property is mapped to the "rec_nm" column of the "rec" table. The recordType property refers to a separately mapped RecordType class, and is associated with Record via the foreign key column rec_type_id. Similarly, each record contains a collection (Set) of fields that refers to a separately mapped Field class, associated with Record via the foreign key column "rec_id".




```
<hibernate-mapping>
    <class name="Record" table="rec"
    mutable="false">

        <id name="id" column="rec_id" type="long">
            <generator class="native"/>
        </id>

        <property name="recordName" type="string"
         column="rec_nm"/>

        <many-to-one name="recordType" column="rec_type_id"
         class="RecordType"/>

        <many-to-one name="iocBoot" column="ioc_boot_id"
         class="IOCBoot"/>

        <set name="fields" inverse="true" lazy="true">
            <key column="rec_id"/>
            <one-to-many class="Field"/>
        </set>
    </class>
</hibernate-mapping>
```

Figure 1. UML Record Description and XML Hibernate Mapping

The runtime behavior of an actual Record object instantiated from the database can be strictly controlled through mapping options and programmatically as well. For example, it would likely not be desirable to have every Record object cause an immediate loading of all associated Field objects in order to satisfy the mapping of fields. As such, the fields mapping is declared "lazy", and therefore the database query to read all the fields will only occur when an iterator of the fields set is requested. This and other means offer considerable control over the extent to which an object graph is instantiated from the underlying database.

The equals method should be implemented only in terms of the business properties of the object. The discipline required to define this for every class reaps many benefits, as it clarifies greatly what the true meaning of the business object is. Additionally, Hibernate contains an object cache that ensures that only one instance of a given unique object will ever exist in memory. The uniqueness is defined by equals. Objects that are part of a collection, such as the set of fields, will behave correctly from a Java collection semantics standpoint as well.

Process Variable Model

The process variable object model was designed to capture the evolution of an EPICS control system configuration over time. The resolution here is the time between reboots of an IOC (or any channel access server). The IOCBoot class is therefore central to the model. Figure 2 shows an abbreviated UML diagram of the process variable class structure. Every time an IOC is rebooted, a crawler script captures the complete configuration of data from the startup script, in particular the data from the dbd and db files. This data becomes a new set of Record, RecordType, Field, and FieldType objects, all associated with that IOCBoot instance. The most recent "snapshot" of the system is identified by retrieving all IOCBoot instances where the currentLoad property is true. Comparisons (or diffs) with past configurations can be done as well using the bootDate property. Since the actual definitions of record types can change, we capture all type information as well as instances. The IOCResource object is used to capture the path to the dbd or db file at the time of the reboot.

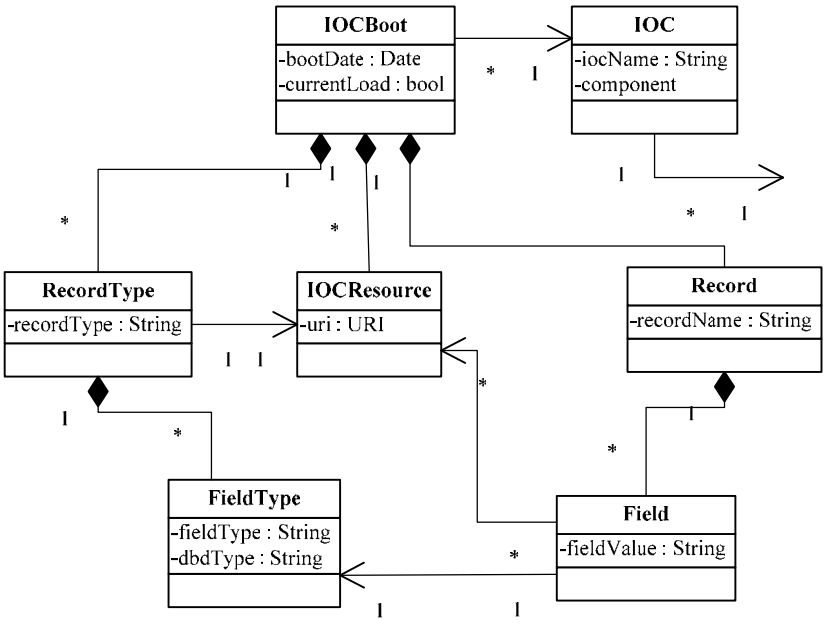


Figure 2. UML Diagram of Process Variable Model

An IOCBoot object is associated with an IOC object. The IOC object is a special case of an IRMIS component and therefore contains a reference to the associated component from the component model. This association is the beginning of the process of mapping hardware-related process variables to the actual ports and pins of components.

Component Model

The component object model captures the configuration of hardware in the accelerator. Figure 3 shows an abbreviated UML diagram of the component class structure. While it is focussed on control system hardware, typically there are many non-control-system components as well. Every Component has a ComponentType. The ComponentType is a generic descriptor, limiting itself to such properties as FormFactor, Manufacturer, Functions, Documents, and responsible Persons. A ComponentType can just as easily represent a VME CPU board as it can a room, rack, or enclosure. The purpose is not to capture large amounts of type-specific data, but to provide a uniform representation. It is important that the core of IRMIS not commit itself to particular uses. Type-specific extensions are readily accommodated as ancillary tables, although the core applications do not use this data. Extensibility and customization are always a consideration in our designs.

A Component by itself has no identity other than perhaps an optional serial number. A component gains its identity through its logical relationships to other components. Each component may contain parent and child relationships to other components. Currently we support housing, control, and power relationships, although this is extensible.

A housing relationship implies physical containment. For example, a CPU is housed in a chassis, which is housed in a rack, which is housed in a room. This is the housing hierarchy. Rather than maintaining a fully realized “location” property for every component, a location can be derived by following the parent housing relationships.

A control relationship implies the logical flow of control and data between components. This logical control flow can be very different from the physical interconnections between components, particularly in a daisy-chain bus. For example, a CPU controls a GPIB controller, which in turn controls a GPIB instrument. This is the control hierarchy.

A power relationship implies the non-control-related supply of AC or DC power to a component. For example, an instrument is powered by a low-voltage converter, which is powered by a line strip, which is powered by a breaker in an AC panel. This is the power hierarchy.

The child relationships of a component may be ordered using the logicalOrder property, and the relationship may also have a descriptive property logicalDescription. For example, a controls bus may

have many addressable nodes. The logicalOrder defines the natural ordering of these nodes, and the logicalDescription is a descriptor significant to the control application engineer. It is often the node number referred to in the INP or OUT field of the associated process variable.

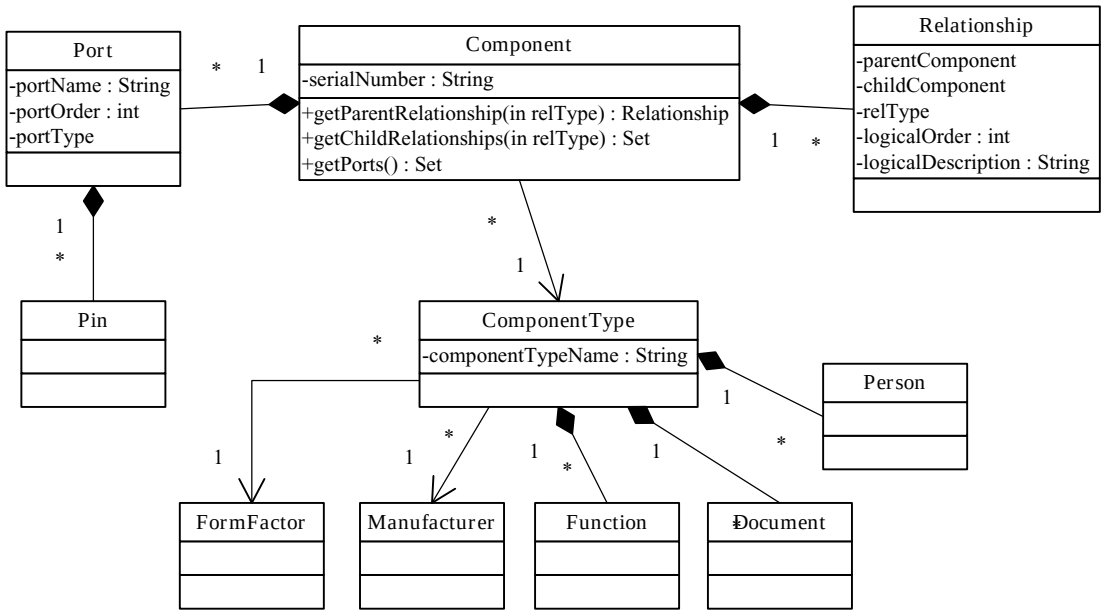


Figure 3. UML Diagram of Component Model

The physical interconnect between components, although not currently completed, is realized through Port and Pin objects. Additional objects will be used to represent conductors and/or cables between these ports and pins. Thus questions about physical interconnections can be answered by following conductors and/or cables between the ports of different components. For example, the reason for a daisy-chain bus failure may not be clear from the logical controls hierarchy, whereas the physical interconnect can provide this data.

SERVICES API

The services API is what the user interacts with to acquire and save IRMIS objects. This API provides methods to instantiate object graphs from the underlying relational data and to save modified object graphs back to the database. The API is stateless and transactional, with the object graph itself retaining the state of a session across multiple transactions. Actual connection and session management with the underlying database is managed by Hibernate and a Data Access Object (DAO) layer, which are not typically used directly.

There are presently three groups of services: PVService, ComponentService, and ComponentTypeService. The methods are all static and can be invoked directly, for example PVService.findCurrentIOCBootList(). This method returns a Java list of IOCBoot objects from which all current process variable names in use can be found through iteration. An abbreviated list of available methods is presented below.

PVService

- List findRecordList(PVSearchParameters searchParams) – returns list of Record that meet constraints given in searchParams.
- List findCompleteRecordBootHistory(Record record) – returns list of IOCBoot in which record appears.
- List findReferringFieldsForRecord(List iocBootList, Record record) – returns list of Field (across all records accessible from iocBootList) which contain a reference to the given record. Effectively returns all links to a given record across the whole EPICS control system.

ComponentService

- List findComponentsByType(ComponentType componentType) – returns list of all Component of the given componentType.
- void saveComponent(Component component) – persists a new component to the database, or updates the data if it exists already. All associated objects of the component will also be saved according to the cascade options in the Hibernate mapping.

ComponentTypeService

- List findComponentTypeList() – returns list of all known ComponentType in the database.
- List filterComponentTypeListByPeerInterface(List componentTypes, ComponentType peerComponentType, boolean peerRequired, boolean peerPresented, String relationshipTypeName) – takes (complete) list of component types, and returns a list of ComponentType that have an interface that matches at least one of the interfaces of the given peerComponentType. This method is used to find candidate components when inserting a new component into the housing, control, or power hierarchies.
- void saveComponentType(ComponentType componentType) – persists a new component type to the database, or updates the data if it exists already.

CONCLUSION

The IRMIS object model and services API provide a significant abstraction for what would otherwise be a daunting programming task. The IRMIS application developer can spend more time solving user requirements and less time dealing with data copying, transaction boundaries, performance, complexity management, and multivendor database support. The IRMIS object graph can be used directly in Swing to drive graphical widgets, without the need to copy data into other structures. Modifications to data anywhere in the object graph can be saved with a single call using a "root" object from the graph. Hibernate automatically detects which objects have been modified, generating SQL transactions for new/modified data only. Hibernate also manages the sequence of cascading writes for associated tables, as well as performing cycle-detection in the object graph.

The semantics of IRMIS itself are complex. We believe that without the support of Hibernate, the object model, and the services API, much of what we are attempting would be largely unmanageable. Our approach does have a nontrivial learning curve. For those accustomed to assembling SQL queries programmatically and managing the result sets explicitly, this approach does require some retraining. We believe the investment in this training pays off greatly in the long run, making complex database-oriented applications more manageable.

Database vendor independence has been maintained, however it is not as simple as a single switch. Our primary development environment is MySQL, but IRMIS has been successfully run against Oracle. The Oracle port required minor modifications to the Data Definition Language files, and performance of some queries varied due to differences in text indexing between MySQL and Oracle.

The reference implementation of IRMIS is reliant on Hibernate and a moderate amount of business logic. It is possible, however, to directly use the relational database through another language. At APS, there are people using PHP to create simple, targeted viewers and editors of IRMIS data. The risk here, and a drawback of increased reliance on business logic, is that direct access to the database could produce sets of database records that do not comply with the soft constraints enforced by the business logic. We have therefore strived to manage that risk by making the schema alone as comprehensible as possible.

REFERENCES

[1] N.D. Arnold, D.A. Dohan, A. Johnson, C. Saunders, "Discovering Process-Variable-To-Signal Relationships in EPICS 3.X and 4.X," ICALEPCS'2005, Geneva, Switzerland, October 2005.

[2] D.A. Dohan, N.D. Arnold, "IRMIS," http://www.aps.anl.gov/epics/irmis.

[3] D.A. Dohan, N.D. Arnold, "The APS Cable Database," ICALEPCS'2005, Geneva, Switzerland, October 2005.