EXPERIENCES: CONFIGURATION MANAGEMENT WITH A GENERIC RDB DATA-MODEL

T. Birke, B. Kuner, B. Franksen BESSY, Germany

Abstract

A new RDB data-model has been introduced at BESSY to enable a more generic approach to store and handle configuration data. Stored data ranges from global hardware-structure and -information through building logical hierarchies to configuration information for monitoring applications as well as signal-level information. This information is used to configure the front-end computers as well as the generic and higher level tools like alarm-handler and archiver.

New applications at BESSY are developed with this generic RDB data-model in mind. First experiences with real-life applications as well as a set of tools for entering, maintenance and retrieval of configuration data are described in this paper.

MOTIVATION

In 1995, when the software- and configuration-foundation was laid at BESSY-II, the decisions made also included a commercial professional grade relational database system (RDB) to hold all configuration data.

The basic approach was to map a specific RDB data-model to each application, while applications were usually separated by device-class (power supplies, vacuum, RF, interlock systems…).

The RDB evolved over the years and now consists of about 250 tables and 150 views storing and providing configuration data in about 20 more or less different data-models. Maintenance of data-structures and -contents took more and more time. Evolving thus changing requirements of applications could not always be weaved into the existing data-models and so lead to patches that again raised maintenance load.

A new RDB was developed with a flexible and generic data-model in mind to get around these limitations and high maintenance load.

A NEW RDB DATA-MODEL

The new RDB data-model basically consists of three tables:

- ***gadgets***
  Entities with nothing but a name
- ***relations***
  named parent-child relationships between *gadgets*
- ***attributes***
  name/value pairs associated with a *gadget*

Using these three tables, any set of directed acyclic graphs of configuration information may be represented. While *gadgets* represent the nodes in the graph, *relations* correspond to the edges and those two represent the structural information of the stored configuration data. The *attributes* finally hold the configuration parameters and data. *Gadgets* may have any number of relations to other *gadgets* as long they can be uniquely identified. Also any number of *attributes* may be associated to a *gadget*. Additionally *relations* also carry a name (which is in turn a *gadget*) resembling some kind of relation-type.

Since the graphs stored in the data-model (gadgets-database) may be of arbitrary depth, recursion or nested loops are necessary to retrieve all connected information. The chosen commercial RDB-system ORACLE has an extension to the SQL-standard called *hierarchical queries (SELECT ... CONNECT BY ... START WITH ...)*. This makes it possible to retrieve any number of nested parent/child relationships with just one query.

The actual implementation of the three core-tables contains a few more columns. For further details and examples, please see [1] and [2].

APPLICATION PROGRAM INTERFACE

Since accessing the tables directly is not only difficult for simple applications, but also dangerous, since the structural information may easily be destroyed, an application program interface (API) is provided to encapsulate basic operations and hide the implementation details. This API is written in PL/SQL.

Bindings to Programming Languages

On top of that API there are several bindings to commonly used programming languages. Currently supported are Java, Perl, Python, Tcl and Haskell.

Java

The Java-binding was the first binding to access the gadgets-database from an application point of view. It was basically developed as an access-layer for the *GadgetBrowser* (see below) and uses JDBC to access the PL/SQL-API.

Perl

The perl-binding is heavily used in a set of command-line-tools (see below) that were developed while building the first application to use the gadgets-database and is based on perl DBI. One main feature of these tools is the possibility to dump the information stored in the gadgets-database to ASCII-files and read these back in again. This was especially helpful during development of both RDB gadgets-database and PL/SQL-API. The ASCII-dumps are, since they are located with the source-code of an application, also kept in a revision control system. This at least opens a possibility to completely switch an application back to a previous version – source-code as well as configuration data.

Python

The Python-binding uses *adodb* to access ORACLE and is only used in some test-programs for now. Nevertheless, this is likely to change, since some few scripts creating configuration files are already written in

Python. The configuration data in this application is already stored in hierarchical python dictionaries that easily map to the gadgets-database.

Haskell

The Haskell-binding opens a whole new and elegant way of programming using the data-store in a functional instead of a procedural way.

APPLICATIONS AND TOOLS

GadgetBrowser

The Java application *GadgetBrowser* is a first approach to have a universal browser and editor for maintenance of information stored in the gadgets-database. It offers access to all basic operations like creating, modifying or deleting *gadgets*, and *relations* and is a simple graphical browser to retrieve the stored information.

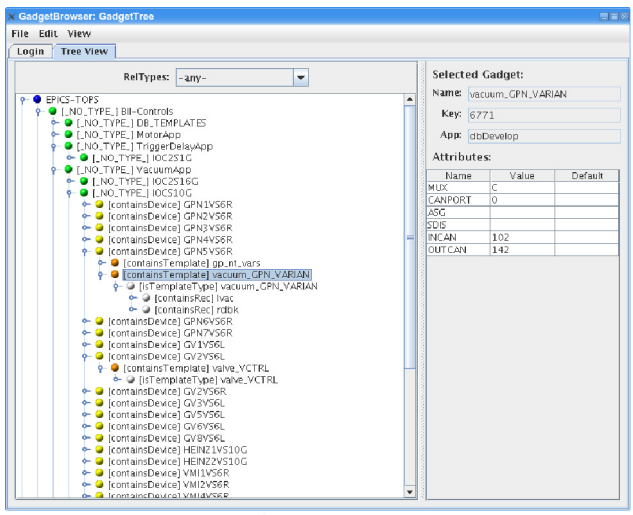

Figure 1: *GadgetBrowser* main screen

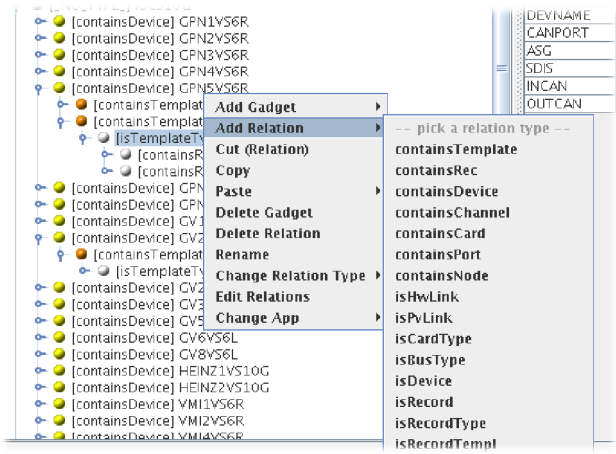

Figure 2: *GadgetBrowser* edit menus

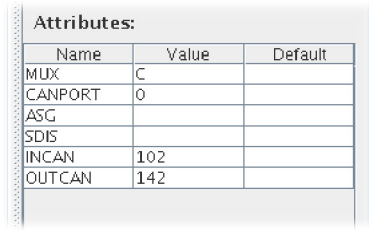

Figure 3: *GadgetBrowser* attribute editor

Perl toolset

A very powerful set of command line tools has been developed using the perl programming language. One basic element of information-representation for these tools is a *path* to a *gadget*. This *path* represents the parent/child relations on a specific path between two *gadgets*. A *path* can indeed be compared to a directory structure on a file-system. So if the gadgets-database contains data as shown in table 1

/EPICS-TOPS/BII-Controls/VacuumApp/IOCS10C/GPN5VS8R

represents a valid path to a device in the example shown in Fig. 1, while

IOCS10C/[containsDevice]/GPN5VS8R

represents a specific relation.

parent	child	relation_type
EPICS-TOPS	BII-Controls	-
BII-Controls	VacuumApp	-
VacuumApp	IOCS10G	-
IOCS10G	GPN5VS8R	containsDevice

Table 1: sample contents of gadgets-database

The core-module in this perl-toolset takes descriptions based on this path-syntax and performs queries or inserts on the gadgets-database. It also logs all actions and thus is able to *undo* all operations and put the gadgets-database back into its former state.

Other scripts in this toolset use the core-functions to retrieve information and – based on conventions – is e.g. able to create runtime DB configuration files containing record-descriptions to run on an IOC (I/O-Controller) running EPICS (Experimental Physics and Industrial Control System).

FIRST STEPS

The first application to use the new RDB Gadgets-database was the Vacuum System Application for the BESSY-II Storage-Ring. Major extensions and changes had to be applied to this Application, so it was chosen to be the first real-world application to move to the gadgets-database.

Most applications for controls-configuration at BESSY use a common approach: Template configuration files are

created from different sources and model the common structure to control e.g. a vacuum valve or a getter pump. These templates contain lots of placeholders where instance-specific information will go (e.g. I/O-address, device-name …). On instantiation, this information is retrieved from the ORACLE database and filled into the template resulting in a complete configuration file. Although this reasonable approach has proven successful, one of the shortcomings was the fact, that information was distributed over heterogeneous places. Some Information was retrieved from the ORACLE database, other information was stored in flat files and some information may even be generated by scripts during build-time (see Fig. 4).

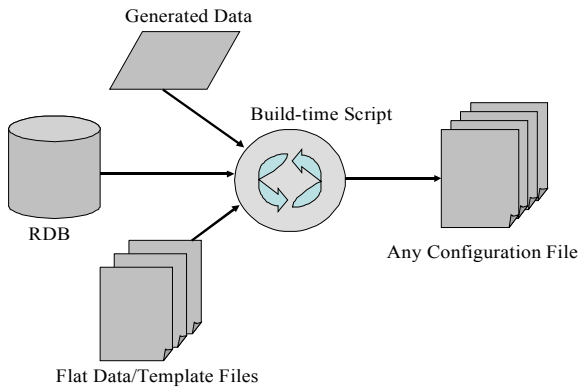

Figure 4: "old" configuration system – still in use

One of the motivations for the new RDB data-model was the ambitious intention to store *all* information in the RDB and put the "intelligent" part of the process in scripts. So basically the information that used to be stored in the RDB had to be restructured to fit into the gadgets-database and the information, that was held in flat files, for the first time had to go into the RDB. This was indeed an instructive process and many improvements were made to the gadgets-database at this time.

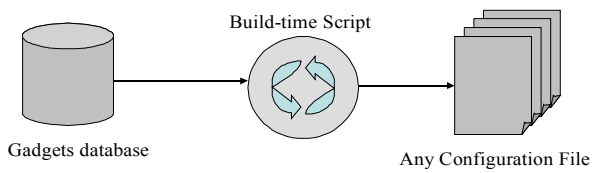

Figure 5: "new" configuration system

The next application to follow was the trigger-delay application handling all timings/delays at BESSY. Currently in preparation-phase to move to the gadgets-database are the insertion-device- and the power-supply-application at the MLS storage ring (see [3]).

All new applications developed at BESSY and for MLS are planned to store their configuration information in the gadgets-database.

A STABLE BASIS

The result of two years of part-time development is a more and more stable basis and the increased experience on how to make use of a highly generic RDB data-model.

The idea of storing any kind of structured information in a generic RDB data model together with proper retrieval and maintenance tools is a powerful foundation, giving full control over the structure of data and the data itself to the developer without any interaction of an RDB-administrator. While in the "old" configuration system (Fig. 4) intelligence was spread amongst RDB, flat files and build-time scripts together with the necessary information, the "new" configuration system strictly separates information from intelligence (what to do with/how to interpret this information). All information is in the RDB data-model and all intelligence is located in the programs and scripts that actually create the needed configuration files.

CONCLUSION

Although re-designing an existing – and for most parts even properly working – system is always a time-consuming and sometimes even painful process. The *new* system has as well to cover all useful and positive elements of the *old* system while also fill the gaps and shortcomings the *old* system never covered well.

This is not always possible and sometimes even well settled and known procedures have to be modified and even seem more complicated at first glance.

In any case – this *filling the gaps* is the main driving force to do the whole switch, and the result will be a configuration management system that is

- Easier to maintain – and thus will reduce workload
- Highly generic – and thus will be easier to adapt to requirement changes
- All from one source – and thus holds all relevant information in one place

REFERENCES

[1] T. Birke, B. Franksen, B. Kuner, R. Lange, P. Laux,, R. Müller, G. Pfeiffer, J. Rahn "Beyond Devices: An Improved RDB Data-Model for Configuration Management", ICALEPCS'05, Geneva, October 2005, PO1.078-7

[2] T. Birke, B. Franksen, B. Kuner, R. Lange, P. Laux,, R. Müller, G. Pfeiffer, J. Rahn "Use Case: Configuration Management with a Generic RDB Data-Model for", ICALEPCS'05, Geneva, October 2005, PO1.079-7

[3] R. Lange, T. Birke, R. Daum, S. Ehlert, D. Faulbaum, B. Franksen, R. Hartmann, B. Kuner, P. Laux, R. Müller, I. Müller, J. Rahn, H. Rüdiger, D. Thorn „Status of the MLS Control System", ICALEPCS'07, Oak Ridge, TN, October 2007, TPPB37