WHERE AND WHAT EXACTLY IS "KNOWLEDGE" IN CONTROL SYSTEMS

M. Plesko*, G. Tkacik, Cosylab and J. Stefan Institute, Ljubljana, Slovenia

Abstract

A straightforward answer would be that Knowledge comprises configuration data and that it is stored in databases. However, this would be only a partial answer. In order to have Knowledge of the control system configuration, it is not sufficient to know only the data themselves, not even the structure of the database tables. Only the interpretation of the data by some application can be seen as true Knowledge. This means that most of the Knowledge is coded into some algorithm or software interface and has therefore little to do with pure configuration data. A simple proof by construction of the above statement is that any application programmer has to read first the documentation, or worse, ask the designer of the database or data server for details. Such Knowledge about data is denoted meta-data. We present a use case showing the need for meta-information in applications and discuss how and where it is handled in different control system designs. In order to be used for true, system independent, generic applications, meta-data must exist in generic machine readable form. In the paper we argue that these meta-data can be described by models, which, although different from system to system, can be reduced to some common types of functionality and described accordingly. After a description of what type of information needs to be contained in meta-data in the case of control systems, we discuss how to obtain meta data from the control system components such as configuration files, RDBMS, etc.. We finish by speculating what kind of new knowledge management and development tools could be built on such basis, bringing thus advantages also to non-generic custom applications and control system management in general.

INTRODUCTION

In this article, we discuss Knowledge in control systems using our standard Cosylab example, a power supply device. We gradually introduce new concepts that describe higher and higher levels of abstraction of a power supply and point out which specific technologies can be used to implement these concepts.

A discussion about the basic concepts of meta-data and models, i.e. *introspection*/*reflection*, is always risky. It tends to be either too formal to be understandable or too poetic to be of any practical use. We will therefore spend a lot of time with examples that illustrate the concepts, just to be sure that the formal definitions do not obscure the flow of reasoning. In this way we hope both to successfully convey the importance of meta-data in control systems and to avoid the two extremes.

*mark.plesko@cosylab.com

USE CASE: A POWER SUPPLY

Definitions

The power supply is a *device* that has the *ability* to be turned on, off and to be reset. It also has a *controllable value* representing the desired current of floating point type; a readback value that denotes the actual current produced by the power supply and the *bit-pattern* value that reports the power supply status (e.g. alarm conditions).

Notice that I am only talking about **concepts** (device, ability to perform an action, value), **relationships** (has) and **attributes** (being of floating point type, being controllable). No mention is made of how these are mapped into any specific design – be it an object oriented accelerator model, a database table or an XML tree. So our reasoning will be true for any control system that controls such a power supply, from channel-based to object-oriented.

Naturally, I have introduced a simplified power supply just as an example, without losing generality in my arguments. The full list of relations and attributes can be significantly longer. All are properly handled in our Abeans libraries [1] and described in more detail in another paper to this conference [2].

Increasing Levels of Abstraction

In order to be able to talk about concepts of knowledge, I have introduced new ones ("concept", "relationship", "attribute"), i.e. I have brought into play concepts on one higher level of abstraction. Let's call a specific power supply, e.g. "PSBEND_M.01" a level 0 or ***instance*** name. It is an instance of device type (or device class for Java and C++ programmers) "PowerSupply", a level 1 or ***type*** name. The noun "PowerSupply" itself is an instance of "entity", which is a level 2 or ***meta-type*** name.

By using such vocabulary we can take a look at how existing control systems handle large numbers of power supplies. Typically, there will be a power supply table application, offering a view of the settings and statuses of a certain subgroup of power supplies.

Abstraction Level 0: Instance Data

The most naive way of creating such an application is to design the GUI and the application logic in whichever programming language, and to put, into the same programming language, also the list of names of all power supplies (a list of level 0, or instance, names). We use the term hardcoding to denote such an approach, because level 0 names are statically enumerated in a language, which is normally used to declare type 1 entities. To be concrete: if you declare a PowerSupply class in Java, create an application that uses it (the power supply table

application) and enumerate all of your 120 power supplies in Java by name, you hardcode.

Abstraction Level 1: Type Data

What if a new power supply is added – should the application code be modified? We see that for reasons of maintainability and flexibility we have to separate level 0 names from level 1 names: Level 1 declarations (types) remain in the code, while level 0 instance listings are moved to a file, or a database. We have thus separated procedures from data, or in other words, introduced the distinction between *procedural* and *declarative* styles. What remains in the code on level 1 are procedures, or rules for manipulating content, which is now level 0. A file containing a list of power supply names in the form of "linac = {PSBEND_M.01, PSBEND_M.02, ..." is now perceived as a declaration of what a "linac" is.

While the traditional approach takes into account (supposedly; let's hypothetically forget about configuration problems) the dynamic nature of instance listings (in other words, content – level 0), it is still prone to the same drawbacks on the type level (structure – level 1, as opposed to content). In order to be able to do anything useful with the list of power supplies, the application uses **application logic** or **algorithms** to display or even control the power supplies. Algorithms presuppose that there is some structure implicit under the power supply name "PSBEND_M.01". For example, my code that reads the current in "PSBEND_M.01" knows that it has to send certain bits over the network which encode "PSBEND_M.01/current" name and it knows it has to expect a single response that has to be interpreted as a coding of a floating point number. In essence, I hardcode the algorithm: I use the knowledge of the structure of power supply in my head and translate it into procedural logic.

Abstraction Level 2: Meta-Data

But even such hardcoding of algorithms can be avoided. We can repeat the same process as in the separation of level 0 and level 1, by introducing and coding level 2, and moving level 1 out of the code or at least into a separate, well-defined and delimited part of the code. Think of using introspection in Java to invoke methods as opposed to simply calling them:

myPS.readCurrent(value)

hardcodes the knowledge about the power supply into the algorithm, while

Class.forName("myPS").newInstance().getMethod("readCurrent").invoke(paramlist, value)

Does not. The latter code snippet contains all information of methods and parameters as strings, which can be conveniently stored in configuration files. Thus, we have data about how to manipulate data, or more generally, data about data, i.e. **meta-data**. Together, data and meta-data form the full knowledge of the control system and by finding a means of describing them formally, we can finally talk about managing Knowledge in a generic way.

One may ask: why introduce the additional level of abstraction? After all, the introspective method execution in the latter code snippet is by far more complex and less clear than the former. A seemingly compelling reason is to be able to cope with the change of device *structure* without any changes in application code, easing maintenance. The price to pay is high, because as such flexible generic (as opposed to traditional) code does not hardcode power supply structure, it not just more complex but also more complicated to design. So is the generalization worth the effort? It is, if we have to write applications such as described in the next section.

BENEFICIAL USES OF META-DATA IN CONTROL SYSTEMS

Treating Knowledge as Configuration Data

Paralleling the benefits gained by separating levels 0 and 1, we again increase both **flexibility** (tolerating addition and removal of content) and **maintainability** (tolerating change in structure) by factoring away level 2 from level 1. Changes in power supply structure are now realized by changing the content in some file or database or XML that says

power supply =

{current, readback, status, on, off, reset}

This is a declarative change as opposed to procedural change and is easier to implement. Moreover, it is seen as a part of a configuration, meaning that configuration management techniques (versioning, backups, centralization) can be applied to it. Maintainability is increased because level 2 structures, such as "entity", "containment", etc., are highly unlikely to change and because level 2 contains such a small number of highly abstract concepts which are common across machines.

Writing Generic Applications

Generic applications get simpler when moving one abstraction level higher, because the number of concepts needed for the complete description of the system on that level will be both **smaller**, and more **universal**. Keeping in mind that the most maintainable code is that which does not exist at all, and taking into account that a generic application is the one which replaces a whole class of applications with similar functions on all existing devices or subsystems, being generic is the most efficient way of making a maintainable product.

Control System Independent Applications

Finally, generic applications can be written such that they use meta-data only. Thus they tolerate changes in structure and content so well that they are **portable** across different machine architectures and control systems, as long as all adhere to the same basic concepts: While my power supply and yours may differ (and thus our traditional power supply table applications would be different and incompatible), we might at least agree that a power supply is an entity, which contains certain actions and properties and that, further, "property" contains a

value that can be read and optionally set, and a set of characteristics; and so on. We will see in this article how it can be done by extracting meta-data from different control system models.

Add a disclaimer: the development efforts are justified only when a company such as Cosylab or a community actually must produce a portable application that runs on several installations. If this is not a requirement, a traditional level 1 application will definitely be the preferred choice.

WHERE ARE META-DATA HIDDEN IN CONTROL SYSTEMS

As it turns out, all control systems deal with meta-data, although they usually don't treat them separately, but rather keep them implicitly in several places.

Meta-Data in Naming Conventions

Take for example a control system that relies heavily on a naming convention approach. What we humans perceive as structure (such as the fact that power supplies have a current, and moreover, even the fact that the power supplies exist as independent entities themselves) is, in the naming convention approach, actually realized on instance level, or level 0. If we stick solely to the information present in the control system, then the control system is a collection of independent channels with names like "PSBEND_M.01:current", "PSBEND_M.01: readback", "PSBEND_M.02:current" and so on. Although our brains see hierarchy in the names there is actually none without our explicit additional knowledge. The point where such additional knowledge gets coded into a program are the routines such as *getDeviceName()* and *getPropertyName()* that split the names at colon ':' signs. The problem with this approach, however, is that it is very easy to be inconsistent: name parsing occurs in many places, colons can incidentally be replaced by semicolons which lead to unpredictable errors, detected only at run-time and so on. In addition, we are not making use of the syntactical apparatus that an object-oriented language is offering.

Meta-Data in Programming Language Objects

On the other hand, if I define a power supply as a C++ or Java class and use object oriented (OO) typing to declare that it is composed of three methods, namely *on()*, *off()*, *reset()* and that it contains by reference three other class instances that represent *current*, *readback*, *status*, I have put more of my human knowledge onto level 1 and less onto level 0. In this scenario, I have to list on level 0 only the instance names of power supplies, no longer of channels. Because I have defined *PowerSupply* to be an OO class, each of my applications that reference this class "knows" automatically that each power supply has a current. The compiler checks at compile time what was before implicit: there are no colon-goes-to-semicolon mistakes. An application programmer does not need tons of paper documentation showing all names and data exchanged when each name is used, but only the list of power supplies and the definition of a single power supply. In addition, the definition can be made in a formal language, such as UML or OMG's IDL. In the case of Java,, there is an additional set of tools that can be used to manipulate these formal definitions, such as *java.lang.reflect* classes, classloaders, javadoc documentation tool, to mention just a few.

ABEANS – A FRAMEWORK THAT DEALS WITH META-DATA

How does one use an object-oriented language such as Java to manage three levels (instance, class, meta-class) instead of two (instance and class)? And what are level 2 concepts that are common to all control systems, what are the relationships between them and what are their attributes? How do we decide whether to put simply more structure into level 1 and less content into level 0, or introduce level 2?

We will show it on a simple example: My goal is to move the power supply structure out from the power supply application to an external data source (XML file, RDBMS, CORBA Interface Repository etc), and to create a set of level 2 concepts that my application will use to *learn* what a power supply is from that external data source. **My new, generic, power supply application will learn what your concept of "PowerSupply" encompasses from an external data source provided by you and will run unchanged on your system as well as it does on mine**. This scheme will work if both you and me use the same level 2 concepts to describe power supplies, because these level 2 things are the ones that remain hardcoded in the generic power supply application.

The Abeans Way

In our effort to produce generic applications we have undertaken a careful study of existing techniques for manipulating meta-data, such as the CORBA Meta-Object Facility[3], W3C's RDF[4] (Resource Description Framework) and XML UML mapping[5]. We concluded that these techniques are too complicated because they have been designed to describe any conceivable level 1 structure, while we can restrict ourselves to control systems. As a result, we have developed our own, reasonably simple yet complete meta-data engine as part of the Abeans framework.

The many level 2 concepts used by Abeans libraries to describe level 1 control system entities are listed and explained in depth in reference [2]. Here, instead of being complete, we will rather explain the main idea and walk through an example.

As we want to deal with meta-data in a generic way, we have separated the way a programmer defines controlled objects (we call that the model) from the way (s)he communicates with the data source (we call that the plug).

There is one plug for each control system API (e.g. EPICS, TINE, ACS, etc.) and one model for each way of

dealing with data (e.g. channel-based, device-oriented, etc.) There is no one-to-one relation between models and plugs: for example, both EPICS and TINE, although having very different plugs, work on the same channel-based model. The application programmer interacts just with models – a plug has been written once and for all, just like a hardware driver.

In between the model and the plug is a thin but crucial layer that treats all data in the same manner, be it a read or write request, an action like on or off, or just a callback waiting for a monitor event to occur. We call that layer the Abeans Engine. It uses a canonical request/response format according to the W3C standard URI (Universal Resource Identifier – similar, but more general than an URL) [6], which can describe any data and interactions within any control system. An Abeans URI name contains schema, authority, hierarchical name and query parts and designates a unique target for request, e.g.:
abeans-ACS://server.cosylab.com/linac/PSBEND_M.01/current/maximum?get

When a request is issued to such target, a response will be expected, carrying the result from the remote system. This is how *all* remote interactions are modelled internally in Abeans. The Abeans meta libraries, through the triplet "URI Name, Descriptor, Naming Context" (see [2] for explanation of this triplet) describe all possible request targets, their names, all possible valid requests that can be issued and all possible responses, including errors, exceptions and connection timeouts.

Be aware that such requests can be more than just simple get/set commands – they can be asynchronous, define callbacks, repeated monitors, etc. What a request can do and what responses are to be expected, the list of parameters, name-value pairs, timeout data, error stack, etc. is all stored in the Descriptor.

All models reduce their interactions with the control system to Abeans Engine requests and responses, which are described by universal level 2 meta objects. When a plug receives an Abeans engine request, it executes the request on whichever actual communication system the Abeans are running, be it ACS CORBA, TINE, EPICS or something else.

A Step-By-Step Walk Through an Abeans Generic Application

It would make little sense to add the Engine as another layer, even though it makes the code cleaner. However, generic applications can exchange data directly through Abeans Engine and obtain the necessary structure through Abeans Meta libraries. To better understand how this is possible, consider the following example procedure:

1. A generic power supply table application starts up on a certain system. Using Abeans, it declares that it will use Abeans Engine directly and accesses the Abeans Directory, a service to the engine, which contains all level names and level 1 types as entities. The Abeans Directory, which contains all meta-data, checks using hardcoded level 2 rules, if a type called "PowerSupply" exists on the system.
2. The Abeans Directory returns the following complex but complete information: "PowerSupply" is indeed an *entity* (a directory entry described by a *descriptor*) and is moreover composed of other entities, which are *on, off, reset, readback, current, status*. The first three are nodes in the directory and listed as *request targets*, meaning that they can be executed and a return value or callback must be expected. Meanwhile, *readback, current* and *status* are composed of more atomic entries, such as dynamic value (with *get* and *set* capabilities which are atomic and are request targets), characteristics (*minimum, maximum, format* etc, with their *get* capabilities that are request targets as well) and so on.
3. In addition, the directory shows that there are 120 instances of "PowerSupply" in the system and lists their names. The Power Supply descriptor also tells that in the Abeans modeling library package there is actually a Java *PowerSupply* class that has 1-to-1 mapping to the directory entries *on, off, reset, current, readback, and status.* In addition, it states that such class can be connected to a process on a remote machine, such that, for example, method *on()* invoked on Java class will actually turn the physical device on.
4. My generic power supply table can now construct its GUI by allocating space for 3 displayer entities that will represent *current, readback* and *status*, using appropriate GUI widgets (*slider* for current that is settable, *gauge* for readback and led panel for *status*). Abeans can help the application by maintaining, as part of meta-information, the records which GUI widget is most suitable for displaying a certain entity. The application, for example, can also produce pop-up menu containing *off, on* and *reset* actions whenever the user right-clicks on a power supply.
5. When, for example, the user selects *on* from such a pop-up menu, the application further analyses the directory descriptor for *on*. It finds out that this is a request target which is invoked by addressing Abeans Engine request to target the URI described by:
abeans-ACS://server.ij.si/linac/PSBEND_M.01/on,
that this request needs no parameters, should be timed to complete within one second, executes asynchronously, returns exactly one *void* response in case of successful execution, produces no linkable transient resource allocations (such as a monitor) and so on. Using this knowledge, the application actually creates a request instance and sends it through Abeans Engine to the plug, turning the power supply on.

Note that the above example works exactly the same, irrespective whether the power supply is a programming object in, say, CORBA, or just a name in a collection of channels. In the latter, the directory would simply report "PSBEND_M.01" to be a *naming context representable entity* (i.e. a name hierarchy) that has no remote

connection available. Instead, *current*, for example, would be *connectable*, i.e. represented by a real remote channel or object, whereas it would be only a property of a remote object in the former. The generic power supply table is obviously not structurally dependent on the power supply class: it will work with power supply that lacks the *readback* property but contains an additional *reversePolarity()* method, for instance.

Extracting Meta-Data for Abeans

Having demonstrated the powerful nature of the generic approach on a specific example, we finally turn to the question of how the directory gets populated by meta-data. After all, it is its quality and organization that guarantees the proper functioning of generic applications. Abeans design *requires* that meta-information is inserted into the directory by the plugs layer. Each plug is free to obtain the data in any way it sees fit. The reason is that only the plug "knows" about the control system and therefore should know where and how to obtain meta-data. – let me enumerate some of the more popular approaches.

- **Naming convention**. If the naming convention is strictly adhered to by all entities in the system, the plug can obtain entity lists and parse entity names to generate hierarchy, as described in section 4.1. Obviously, the strictness of the naming convention is proportional to the amount of data that can be automatically extracted by the plug and placed into the directory. All exceptions have to be hardcoded.
- **Class declarations**. If there is structure present on level 1 in the model as Java class or CORBA IDL definitions, then introspection / reflection capabilities of Java or CORBA Interface Repository can be used in conjunction with design pattern rules (similar to those in Java Beans) to extract meta information from class definitions.
- **Structure database.** Sometimes meta-data are stored separately in an SQL database or XML structure file. The plug can parse such resource and populate the directory. Often, however, such resources were designed for a specific purpose and lack all the data that the directory requires. In this case information in a database must either be supplemented with additional hardcoded information, name pattern recognition and so on.

The described sources of meta-data are often incomplete and have to be combined to fully populate the directory. Abeans sets as their goal the interoperability of generic applications and therefore requires that the plugs provide such data. In fact, the problem of re-writing traditional applications during porting is now reduced to providing meta-data with level 2 Abeans structure by the plug (communication system driver), while the application remains the same, along with its GUI representation.

CONCLUSIONS

Let's conclude with future possibilities of the meta-data approach. Should we go even one step beyond meta-data and introduce abstraction layer 3, whatever that may be? It is actually not necessary, as the meta-data concept is powerful enough to apply it on itself, i.e. to use meta-data to describe meta-data. We actually plan to describe Abeans with their own meta-model, i.e. make Abeans controllable as if they were a control system from within Abeans. This would offer intriguing possibilities of making design-time and run-time equal and would enable the applications to be constructed, modified and tested dynamically on a running system. A further line of thought that offers great potential benefits is the analysis of data types, data transformations and views that exist in control systems: the directory could contain instructions on possible visual and textual representations of the data, adding some "common sense" to completely generic browsers (such as the Abeans Explorer [7]). Consequently the program would "know" if a certain array represents a time series, a profile, how it is indexed, if it makes sense to speak about a single value of the array and so on. The user would benefit because many data display and analysis problems could be solved only once and would behave in the same way among applications. Concepts known from office suites, such as clipboard, drag&drop, report creation and so on start to make sense in this context. Using those tools, any conventional application could be instantiated at run-time from a simple configuration file. The problem of renaming and moving/removing controlled devices to/from applications would be trivially managed from a central configuration database. Rolling back to previous versions of applications would mean just to replace the configuration file with an old version.

REFERENCES

[1] I. Verstovsek et al., "Abeans: Application Development Framework for Java", this conference.
[2] G. Tkacik, M. Plesko "A Reflection on Introspection", this conference.
[3] Petr Hnetynka, http://nenya.ms.mff.cuni.cz/teaching/seminars/2003-04-23-Hnetynka-MDA.pdf
[4] O. Lassila et al., http://www.w3.org/TR/REC-rdf-syntax
[5] Object Management Group, XML Metadata Interchange, http://www.omg.org
[6] T. Berners-Lee et al., RFC 2396, http://www.ietf.org/rfc/rfc2396.txt
[7] I. Verstovsek, EPICS spring collaboration meeting 2003, Abingdon, UK