Proc. IFAC World Congress, Sydney, Australia, 18-23 July 1993. ©IFAC

DATA-BASE MANAGEMENT GUIDELINES FOR COMPUTER-AIDED CONTROL ENGINEERING

James H. Taylor † and Georg Grübel ‡

†Odyssey Research Associates, Ithaca, New York 14850-1326 USA

‡Institut für Dynamik der Flugsysteme, D-8031 Oberpfaffenhofen, Wessling GERMANY

Abstract. Rigorous and supportive engineering data-base management (EDBM) was accorded little serious consideration in computer-aided control engineering (CACE) software development prior to the mid 1980s. By then, CACE environments had become very powerful and extensive, and thus the need for keeping track of models, analysis results, control system designs, and validation study results had become increasingly evident. Two motivations spurred an interest in EDBM: it may be very costly to neglect this task, and a well-designed EDBM support can also increase productivity in the analysis and design process.

Software for EDBM should be designed as an integral part of a CACE environment. Here the features of such an integrated EDBM system will be proposed and described. In particular, we will discuss EDBM organization and features that improve CACE productivity.

Key Words. Database management systems, computer-aided design (CAD), control system design, system documentation, software tools.

1 INTRODUCTION

There has been substantial progress made in the past decades in the development of software for computer-aided control engineering (CACE). Primary emphasis has been placed on core CACE activities, e.g., nonlinear simulation, linear analysis and design in the frequency and time domains, and model identification. In the 1980s the spectrum of CACE broadened to include, for example, advanced modeling and autocode generation. One important area has received little attention until quite recently: engineering data-base management (EDBM).

As CACE environments become more powerful and extensive in terms of systems analysis and design capabilities, the need for keeping track of the models, analysis results, control system designs, and validation study results becomes more pressing. In a real industrial control system design project, the large number of files generated in the complete design cycle and the relations among these files can be very difficult to comprehend and manage using manual means which require strong discipline and considerable added expense. On the other hand, it may be very costly (in terms of future errors or need for re-engineering) to neglect this activity. Building an EDBM system into the CACE environment solves that problem. As a further incentive for EDBM, there are also very positive reasons to consider its use - primarily, these are in the areas of improved documentation of systems and increased productivity in the analysis and design process.

Tools for EDBM should be designed as an integral part of a CACE software system. In other words, the control engineer should be able to store and retrieve data and models interactively within the CACE environment, with little or no additional burden. In the sections that follow the features of such an integrated EDBM system will be proposed, motivated and described. This material is based on concepts from several projects in which EDBM has been addressed in a substantial way, namely ANDECS (Analysis and Design of Controlled Systems) [1] and MEAD (Multi-disciplinary Expert-aided Analysis and Design) [2]. We will discuss two issues: EDBM *organization,* and EDBM features that *enhance CACE productivity*; for the former we will use the MEAD organizational model and for the latter we will use the advanced concepts from ANDECS. We note that these considerations are also receiving attention in the Open CACE initiative [3] under "support services", in recognition of the importance of such functionality.

One important related area will not be discussed, namely data structures – we take the data produced and consumed by the various CACE components of the system "as is", with data translation capabilities built-in, as needed. Recent work in developing and defining data structures for CACE may be found in [4, 5].

2 DATA-BASE ORGANIZATION

The organization of a CACE data base is key to its utility and to its "user-friendliness". Organizational issues include accessibility, maintainability of critical relations, and documentability. Such a system can be described by defining the *schema* of the data base and by illustrating the implementation of the corresponding EDBM in terms of a supporting *command language.* The description that follows is based primarily on MEAD's EDBM system [6].

2.1 Data-base Organization – Schema

In terms of organization, we propose that such a data base should be organized in a hierarchical framework having at least the levels *Project*, *Model*, *Attribute*, and *Element* [6], as illustrated in Fig. 1. Additional levels might be introduced for large multi-user systems (e.g., *Subproject*). The level *Project* accommodates the control engineer working on several projects, providing a convenient separation of workspace. Thus, for example, a flight-control engineer might have projects **F-18**, **G-xx**, etc. while a generalist may set up projects **Flight_Ctrl**, **Process_Ctrl**, and so on.

Within a project, models (e.g., of plants, control systems, etc.) are the main focus. Each *Model* has two attributes, a *Description*, i.e., a catalog of the fundamental characteristics of the model (linear or nonlinear, continuous- or discrete-time or mixed, etc.) plus a list of the components that comprise the model and a definition of the component interconnections[1], and a *Result_set*, i.e., the set of analysis and design results obtained using the model. Elements of a *Description* characterize the system in terms of component models (e.g., representations of a plant, compensator, sensor, etc.); elements of a *Result_set* include any result generated with the corresponding model, such as a time-history, frequency response, etc. *plus* a record of the previous set-up activity (e.g., setting initial conditions and parameter values, defining input signals, selecting algorithm options). This latter portion of the result_set is called the *Condition_spec*, i.e., a specification of the conditions used in generating the result; note that this data structure plays an important role in Sections 3.2 and 3.3 (retracing analysis and design experiments).

[1]Note that our discussion of a Model Description assumes that models are composed by suitably connecting a set of components – dealing with monolithic models would be simpler but less flexible.

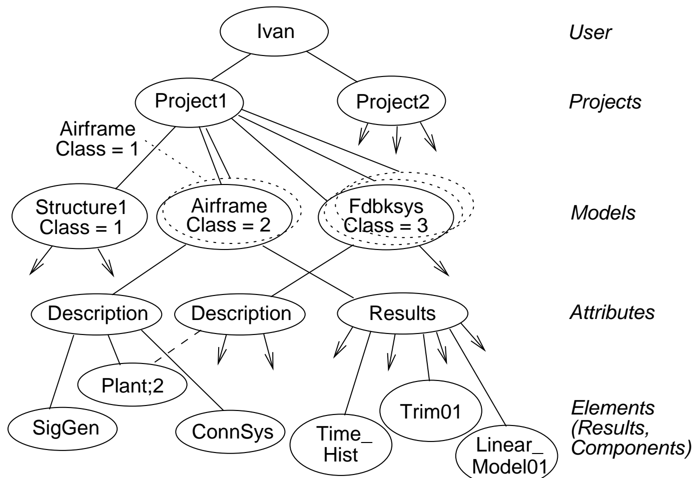

Figure 1: Data-Base Hierarchy for CACE

Given the hierarchical framework outlined above, there are several considerations and features that are vital for maintaining the integrity and documentability of the data base; these include [6]:

1. The use of a Model *version control scheme* is essential, so that system models can evolve and results obtained with each "version" of the model can be unambiguously associated with that version. Given that models are composed of components, this would actually be implemented at two levels, as illustrated in Fig. 1: Each system **Component** has a **version number** which is incremented with every change, and every **Model** instance composed of a set of component versions is assigned a **class number**. Each result is thus associated uniquely with its **Model class**.
2. The use of a *Reference* to relate each subsidiary model (such as a result from linearization, model reduction, model identification, etc.) to the "parent" or model from which it was produced is also vital. For example, a linearization of the nonlinear system **NL_Plant** produces both a Result and a potential Model, e.g., **Lin_Plant**; in MEAD [2] this potentiality is realized by *modelizing* the result, by which it is meant that it is installed in the data base as a

Model, ready to use for whatever purpose. If Lin_Plant were modelized, then it can be analyzed and have a Result_set of its own. References allow such models and their origins to be maintained with integrity; without this feature, the value of each subsidiary model would be jeopardized.

3. The use of *links* to maintain a Component that may be used in a number of Models greatly facilitates its maintenance. For example, the component **Plant** may be used in the models **Bare_Plant**, **Open_Loop**, and **NL_Fdbk_Sys**; the component is actually stored and maintained under one model (probably Bare_Plant, called its "home") and the Descriptions of the other models simply link to its "home" as the source for its file. This eliminates the need to keep separate copies of component Plant for each model, making its maintenance much easier and more secure.

The hierarchical data-base schema and the three mechanisms outlined above for maintaining the critical relationships among the data elements provide the required foundation for the productivity enhancements presented in Section 3.

2.2 Data-base Organization: Commands

The most succinct way to specify the implementation of an EDBM that conforms to the schema outlined above is to delineate a corresponding command language [2]:

1. **Model Building Commands** – Models are created by defining components and specifying their connections, or by "modelizing" a result:

   $>$ Model_create(name, type, form) – this command initiates model building.

   $>$ Component_create(name, type, file_id) – the data structure in the specified file (nonlinear dynamic equations, $\{A, B, C, D\}$ data, etc.) is to be used in defining the named component of specified type.

   $>$ Component_lib_fetch(component_id, (link | copy)) – in MEAD there is a Project called Library in which components are stored; this command allows the modeler to copy OR link with these components. The details of this command would depend on the implementation of a library.

   $>$ Component_copy(name, component_id) – copy a component from any source for use in a model (there is no link – it will be maintained separately).

   $>$ Component_link(name, component_id) – the named component in the model being built will be linked with the specified component in an existing model.

   $>$ Model_connect(type | file_id) – the model is to be configured in a standard way (type = parallel, series, feedback_system ...) OR the data in the specified file defines the configuration of the system.

   $>$ Modelize(name) – install a result (linearization etc.) in the data base as a one-component model of given name.

2. **Model Management Commands** – Models may be "configured" (loaded into a CACE package and readied for use) or modified:

   $>$ Configure(model_id) – defined above.

   $>$ Model_edit(model_id) – modify a model's description (may not be needed if the connection is regarded as another component, as in MEAD).

   $>$ Component_edit(component_id) – self-explanatory.

   $>$ Component_replace(component_id, file_id) – the previous definition of the component is completely replaced by the data structure in the specified file.

   $>$ Component_transfer(component_id, model-_id) – move a linked component to another "home" model (would be necessary if the old home model is going to be deleted).

   $>$ Create_new_class(model_name) – one or more revised components are assembled to create a new model class.

   $>$ Delete_class(model_id) – eliminate a model class (usually done if it is broken or otherwise useless; associated results (if any) are discarded along with the class.

   $>$ Delete_model(model_name) – self-explanatory.

3. **Data-base Management Commands** – Elements of the data base may be catalogued, fetched, deleted or annotated (supplied with notes):

   $>$ Projects – produce a listing of all projects in

[2] This is not the MEAD command language – it is untested but conveys the needed functionality.

the user's data base.

> Set_project(project_name) – specify the project in which the user will work (use models, produce results).

> Models(project_name) – produce a listing of all models in the specified project.

> Description(model_id) – produce a listing of all components that make up the specified model.

> Results(model_id) – produce a listing of all results generated with the specified model.

> Fetch_model(model_id) – self-explanatory.

> Fetch_component(component_id) – self-explanatory.

> Fetch_result(result_id) – self-explanatory.

> Fetch_condition_spec(result_id) – determine the set-up conditions used in generating the specified result. It is advantageous to store this information as the original command set, so it can be used as a macro (e.g., to generate an analogous result with a new model class).

> Fetch_link(component_id) – determine where a component has its home, or if this is the home, determine where else it is used.

> Fetch_reference(result_id | component_id) – determine the home model of a modelized result (if argument = result_id), OR its parent (if argument = component_id).

> Delete_project(project_name) – self-explanatory.

> Delete_result(result_id) – self-explanatory.

> Annotate_project(project_name) – Store comments and observations at the project level. Each note insertion or modification is time stamped.

> Annotate_model(model_name) – as above; comments might include information about changes from class to class.

> Annotate_component(component_name) – as above; comments might include information about changes from version to version.

> Annotate_result(result_id) – Store comments and observations about a specific result.

Note that the above command set makes an important distinction between entities with the suffix "_name" and those with "_id". The former denotes a simple name (e.g., **F-18**) while the latter is a unique data-base pointer, i.e., **model_id = project_name + model_name + class_number**, **component_id = project_name + model_name + component_name + version_number**, and **result_id = project_name + model_name + class_number + result_name** .

There are other needs and features that could be introduced, particularly in the area of support for larger efforts with multi-user EDBM requirements. These are discussed briefly in [7].

3 ENHANCED CACE PRODUCTIVITY

In terms of enhanced productivity, there are several ways in which a well-designed EDBM can achieve this goal [8, 9]: It can –

1. provide an object-oriented paradigm for dealing with models and results;
2. support widget, tool and package integration;
3. permit application/task decomposition; and
4. enable the definition, execution and retrieval of design-history databases by structuring retraceable "design chains".

Besides MEAD [2, 6, 7, 9], ANDECS [1, 8, 10] is another major CACE system with a fully integrated engineering database. The ANDECS technology emphasizes engineering-efficient 'computational experimentation' by supporting the exploratory and iterative nature of control-dynamics analysis and design. In addition to control-engineering efficiency, concurrency with adjacent engineering-dynamics disciplines is of prime concern. The description that follows is based primarily on ANDECS [1], as that package is more advanced in these areas.

3.1 Support of an Object-oriented Data Paradigm

Parnas' data abstraction [11] via Abstract Data Types (ADTs) is a technique to decompose complex computational tasks. Safe and efficient data handling and data processing can be supported by a database system which recognizes appropriate ADTs and provides computationally optimized ADT handling tools.

The RSYST engineering database used in ANDECS supports generic ADTs such as: 1-, 2- and 3-dimensional arrays of integers, real- and double-precision numbers; parameter sets of numeric and

symbolic values; text; and pixel-images. The database access and storing mechanism is optimized to work efficiently with large data objects, such as numeric arrays of time series.

An object-oriented database with a class concept and an inheritance mechanism provides a means to formally define complex data objects composed of generic data types. This is used in ANDECS to formally define Control Data Objects (CDOs) [12] which are basic data structures in CACE: signals, nonlinear input/output dynamic systems, linear dynamic systems in state-space form, transfer function matrices, frequency responses, pole/zero patterns, etc.

While CDOs are just data structures encapsulated by the database class schema, one could go one step further in defining computational objects which encapsulate a CDO together with the system theoretic methods possible to operate on that particular data object. In ANDECS, this 'early binding' approach is not pursued, however. Rather, the database support of ADTs and CDOs is exploited by a versatile modules concept which is based on the concept of 'late binding' as advocated by Cox [13]. This allows an evolutionary development and set up of rather complex computational chains and loops. This is dealt with next.

3.2 Support for Widget, Tool and Package Integration

In CACE-computational experimentation, rather complex systems of interactively controlled computational chains and loops have to be composed from a broad variety of simulation, analysis, order-reduction synthesis, control synthesis, and optimization methods. To compose such experimentation set-ups from more basic computational elements, the concept of 'tool abstraction' [14], which is the complement of 'data abstraction' [11], provides utmost configuration freedom and reusability of proven algorithmic components. Computational tasks are decomposed into suitably simple function modules ("toolies") which communicate their input/output data only via the database. Since by this mechanism no direct module-to-module communication is possible, this allows the configurability of computational chains by side-effect-free modules. Thereby the database serves the purpose of syntactic integration in 'late binding' of complex application tasks. This constitutes the 'database-centered' modular software architecture of ANDECS.

Note, that here it is important that the database is optimized for efficient access of engineering data structures, in particular complex data objects which include large arrays of numerical data. In ANDECS, such an engineering database can also be kept resident in main memory to reduce the access overhead, e.g. when used within an optimization loop.

To associate the results produced by a function module to the originating CDOs and the chosen algorithmic data, a hierarchical database is best suited. With the 'dialog-generator paradigm' proposed in [8] then an automatically evolving 'producer-oriented' database structure can be evoked when executing a function module. The simulation module ANDECS_DSSIM [15] is an example thereof, where all producing data (method, input functions, timegrid, ...) are automatically stored together with the simulation results below a node of the hierarchical database, where the system model CDO is stored. This allows complete and simple-to-do retracing of any simulation run.

A database-centered software architecture also facilitates the integration of different proprietary tool packages within an application: Since the database constitutes a neutral data communication platform through the generic data structures it supports, each external package needs just one data interface to the database. Therefore the number of interfaces necessary for n packages involved, is only of order(n) instead of order(n^2) if the packages were allowed to communicate mutually among each other. Moreover, if the same database system with the same data structure is used in different application systems, the package/database interfaces are reusable, which is a step in the direction of 'open application systems'.

3.3 Support of Application/Task Separation

A gross decomposition in the development and use of application programs is a task separation into three parts: pre-processing, main-processing, and post-processing. This is also a task separation where most likely different packages are involved. In particular, plant modeling by domain specific modeling tools may be considered as preprocessing for CACE analysis-, synthesis-, and simulation main processing. Result visualization, -export and -documentation is often considered as postprocessing since one operates on result data of the previous computations. Again, a database can help to stratify the sequential input/output structuring of such a task decomposition. Specifically, if more than one database realization can be used in one application,

for each task subgrouping one database can be used and all databases can be operated together when group results have to be integrated. This facilitates transaction management in collaborative work. In ANDECS eight databases can be used concurrently within one application system.

3.4 Support of Design History Retraceability

A more detailed, method-oriented CACE-task decomposition is supported by ANDECS for the task of 'design experimentation'. Design experimenting embodies (I) model experiments, i.e. exploring what is a suited design dynamics model, (J) method experiments, i.e. exploring the behavior of a chosen model via various analysis methods in state- and frequency domain or via time simulation, and (K) parametric experiments, i.e. doing parametric iterations either by hand or by an optimizer to explore the best possible attributes of design instances.

The variability of possible design decisions on the strata I, J, K; and the amount of computational data to get the proper indicators for these decisions, make it necessary to support the designer in

- formally structuring the decision process,
- automatically recording the various design steps and the pertinent data,
- allowing to compare any design outcomes by making the necessary indicator data easily recoverable,
- allowing backtracing and branching of the design process.

The 3 decision strata I, J, K form a hierarchy. Hence a hierarchical database is best suited to record and to show the design history [16]:

By means of the *design indices* I, J, K not only the different design steps can be distinguished, but it is also possible to make evolve the data structure automatically as design proceeds. For this, simple rules uniquely map the sequence of design decisions onto the design-data structure:

1. decision on design model:

$$I, J, K \longrightarrow I+1, 1, 1, \text{ if } K \neq 1 \ . \qquad (1)$$

   This means that in case of model modifications, the design index I is augmented by 1, whereas J and K are reset to 1. However, the indices are only changed, when $K > 1$,

2. decision on performance indices (methods):

$$I, J, K \longrightarrow I, J+1, 1, \text{ if } K \neq 1 \ . \qquad (2)$$

   The evolution of the design index J is in accordance with the first rule. The index I of the node models now remains unchanged, because the model specifications are not concerned by this design decision.

3. iteration in parameters:

$$I, J, K \longrightarrow I, J, K+1 \ . \qquad (3)$$

   After the design parameters are specified, a synthesis step for a parameter change can be performed. The resulting synthesis data are stored below the node design. $K + 1$.

This data structure gives the desired decision support, in that all design steps are completely recorded, and immediate backtracing is possible. Branching of the design process can be done by making any existing design I, J, K to the first branch of a new design-data structure. From the stored data all analysis data involved can be easily recomputed, since any synthesis step can be reproduced. Hence a comparison between any design outcomes is possible.

Hence a hierarchical database can serve the integration of a parametric design process on syntactic, semantic and method level.

4 CONCLUSION

Several recent computer-aided control engineering (CACE) projects have demonstrated the need for engineering data-base management (EDBM) in this area[1, 2]. They have also provided a detailed analysis of requirements for functionality (accessibility, maintainability, integrity, documentability) and for productivity enhancement. Such a system would appear to be especially important for Open CACE Software[3]. It is our hope that the discussions and specific details provided above will encourage the wide-spread and standardized introduction of EDBM into existing and future CACE environments.

References

[1] Grübel, G. (1992). ANDECS: a Concurrent Control Engineering Project. *Proc. 1992 IEEE Symposium of Computer Aided Control System Design*, 17-19 March, Napa, California, pp. 37-46.

[2] Taylor, J. H., C. M. Rimvall and H. A. Sutherland. (1993). Computer-Aided Control Engineering Environments. A chapter in *CAD for Control Systems*, D. A. Linkens, Ed. Marcel Dekker, New York. See also:
Computer-aided Control System Environments: Architecture, User Interface, Data-base Management and Expert Aiding. *Prepr. 11th IFAC World Congress*, 13-17 August 1990, Tallinn Estonia.

[3] Barker, H. A., M. Chen, P. W. Grant, C. P. Jobling, and P. Townsend. (1992). An Open Architecture for Computer-Assisted Control Engineering. Presented at *IEEE Symposium on CACSD*, 17-19 March, Napa, California.

[4] Maciejowski, J. M. (1988). Data Structures and Software Tools for CAD of Control Systems: A Survey. Plenary Lecture, *Prepr. 4th IFAC Symp. on CAD in Control Systems '88*, 23-25 August, Beijing, PR China, pp. 27-38.

[5] Harbison-Briggs, K. A., H.-D. Joos, and J. M. Maciejowski. (1993). EXPRESS Data Definitions for Control Engineering. *Prepr. 12th IFAC World Congress*, 18-23 July, Sydney, Australia.

[6] Taylor, J. H. (1993). Data-Base Management for Computer-Aided Control Engineering. A chapter in *Recent Advances in Computer-Aided Control Systems Engineering*, M. Jamshidi and C. J. Herget, Eds. Elsevier, Amsterdam. See also:
Taylor, J. H., K-H Nieh and P. A. Mroz. (1988). A Data-Base Management Scheme for Computer-Aided Control Engineering. *Proc. American Control Conference*, 15-17 June, Atlanta, Georgia, pp. 719-724.

[7] Taylor, J. H., C. M. Rimvall and H. A. Sutherland. (1991). Future Developments in Modern Environments for CADCS. *Prepr. 5th IFAC/IMACS Symp. on CAD in Control Systems '91*, 15-17 July, Swansea, Wales, pp. 51-62.

[8] Gaus, N. and G. Grübel. (1991). How to Tie the Man-Machine Dialog to a Project-Data Structure in Concurrent Control Engineering. *Prepr. 5th IFAC/IMACS Symp. on CAD in Control Systems '91*, 15-17 July, Swansea, Wales, pp. 127-131.

[9] Mroz, P. A., P. McKeehen and J. H. Taylor. (1988). An Interface for Computer-Aided Control Engineering Based on an Engineering Data-Base Manager. *Proc. American Control Conference*, June 15-17, Atlanta, Georgia, pp. 725-730.

[10] Grübel, G., H.-D. Joos, R. Finsterwalder and M. Otter. (1993). The ANDECS design environment for control engineering. *Prepr. 12th IFAC World Congress*, 18-23 July, Sydney, Australia, We-E-3.

[11] Parnas, D. L. (1972). On the Criteria to be Used in Decomposing Systems into Modules. *Comm. ACM*, **15**, No. 12, pp. 1053-1058.

[12] Joos, H.-D. and M. Otter. (1991). Control Engineering Data Structure for Concurrent Engineering. *Prepr. 5th IFAC/IMACS Symp. on CAD in Control Systems*, 15-17 July, Swansea, Wales, pp. 107-112.

[13] Cox, B. J. (1990). Planning the Software Industrial Revolution. *IEEE Software Magazine*, Nov. 1990, pp. 25-33.

[14] Garlan, D., G. E. Kaiser and D. Notkin. (1992). Using Tool Abstraction to Compose Systems. *IEEE Computer*, June 1992, pp. 30-38.

[15] Gaus. N. and M. Otter. (1991). Dynamic Simulation in Concurrent Control Engineering. *Prepr. 5th IFAC/IMACS Symp. on CAD in Control Systems*, 15-17 July, Swansea, Wales, pp. 123-126.

[16] Joos, H.-D. (1991). Automatic Evolution of a Decision-Supporting Design-Project Data-Base in Concurrent Control Engineering. *Prepr. 5th IFAC/IMACS Symp. on CAD in Control Systems '91*, 15-17 July, Swansea, Wales, pp. 113-117.