STUDIES IN AUTOMATION AND CONTROL 9

RECENT ADVANCES IN COMPUTER-AIDED CONTROL SYSTEMS ENGINEERING

Edited by

M. JAMSHIDI
University of New Mexico
Albuquerque, NM, USA

and

C. J. HERGET
Lawrence Livermore National Laboratory
Livermore, CA, USA

ELSEVIER

Amsterdam – London – New York – Tokyo 1992

ELSEVIER SCIENCE PUBLISHERS B.V.
Sara Burgerhartstraat 25
P.O. Box 211,1000 AE Amsterdam, The Netherlands

Library of Congress Cataloging-in-Publication Data

```
Recent advances in computer-aided control systems engineering / edited
  by M. Jamshidi and C.J. Herget.
       p.    cm. -- (Studies in automation and control ; v. 9)
    Includes bibliographical references and index.
    ISBN 0-444-89255-9
    1. Automatic control.  2. Computer-aided engineering.
  I. Jamshidi, Mohammad.  II. Herget, Charles J.  III. Series.
  TJ213.R435  1992
  629.8'9--dc20                                              92-30695
                                                                 CIP
```

ISBN: 0-444-89255-9 (Vol. 9)
ISBN: 0-444-41792-3 (Series)

Recent Advances in Computer-Aided Control Systems Engineering
M. Jamshidi and C.J. Herget (Editors)
© 1992 Elsevier Science Publishers B.V. All rights reserved.

Computer–aided design of intelligent controllers: challenge of the nineties[†]

F.E. Cellier, L.C. Schooley, M.K. Sundareshan, and B.P. Zeigler

Department of Electrical and Computer Engineering, University of Arizona, Tucson, Arizona 85721, U.S.A.

E_mail: Cellier@ECE.Arizona.Edu

Abstract

Major developments in Computer–Aided Control System Design (CACSD) software are briefly reviewed. It is demonstrated that today's CACSD tools are insufficient for supporting the design of intelligent controllers. Major components of an intelligent control architecture are presented, and it is outlined how these relate to CACSD software design issues. It is suggested that the development of adequate CACSD tools for intelligent controller design represents a true challenge for the nineties.

1. INTRODUCTION

In the sixties and early seventies, CACSD researchers concerned themselves mostly with the efficient design of control algorithms. It was felt that the numerical algorithms that were needed to *design* good controllers were quite delicate, whereas the resulting control algorithms themselves and the implementation of these control strategies, once designed, were fairly trivial undertakings. These research efforts resulted in a set of more or less decently maintained libraries of CACSD Fortran subroutines [8]. The users of these subroutines concerned themselves only with the subroutine interfaces, and had often little to no understanding of the computations that went on inside these routines.

In the later seventies, the focus of CACSD research shifted to questions of user interface design. Initially, the purpose of these renewed efforts was to make the previously developed library routines easier to apply. The libraries had become quite bulky, and the development of new application programs calling upon one or several of these library routines was a slow process. The efforts were not driven by a desire for improved flexibility. Their only purpose was to protect the users from having to learn how to call the library routines, which often had a formidable list of user–specifiable parameters.

[†] Research supported by NASA/University of Arizona Space Engineering Research Center for the Utilization of Local Planetary Resources (SERC/CULPR).

These CACSD tools were menu–driven, utterly inflexible, and therefore limited in their scope to the design of controllers for textbook applications [1].

A major breakthrough was achieved by Cleve Moler in 1980 with the design of Matlab [27]. The focus of Matlab was no longer that of protecting the user from having to understand control algorithms. In fact, Matlab hadn't originally been designed for control engineers at all. Matlab is a very intuitive high–level language for linear matrix algebra. Matlab protects the user from having to program in Fortran, and from having to call Linpack [14] and Eispack [18] routines. The reason why Matlab became immediately popular among control engineers was the fact that most CACSD subroutines made extensive use of Linpack and Eispack routines. By porting these algorithms from Fortran to Matlab, the previously obscure CACSD algorithms became at once very easy to read and understand, write and enhance, and ultimately maintain. As Alan Laub once put it: "Control engineers don't have problems with designing controllers, they have problems with computing eigenvalues" (private communication). The previous generations of CACSD tools had offered monolithic building blocks with too coarse a granularity, with the effect that these tools consisted of large numbers of partially overlapping and inflexible building blocks.

The potential value of the Matlab approach to CACSD was quickly recognized, and, in the sequel, several CACSD tools were designed on the basis of either the Matlab software itself [22,26,36,38] or at least the Matlab language definition [17,32]. Three of these tools (CTRL–C, Matrix$_x$), and ProMatlab became commercially available around 1985, and they still define the state–of–the–art in CACSD software. These tools offer compact high–level matrix manipulation languages to describe control system design algorithms in, and especially ProMatlab has taken a very clear stand on standards relating to *toolbox design* issues. While ProMatlab provides for a suitable environment to describe control algorithms in, ProMatlab's toolboxes assume the place of the earlier CACSD subroutine libraries. They encode actual CACSD algorithms for various application areas. However, contrary to the obscure and error–prone Fortran subroutines of the previous era, the ProMatlab toolboxes are themselves compactly coded in ProMatlab, and are easily understandable, maintainable, and extendable.

Three trends have dictated the CACSD software design over the past five years:

1. All three of the aforementioned tools (CTRL–C, Matrix$_x$), and ProMatlab have been merged with nonlinear simulation programs (Model–C, System–Build, and SimuLab) for the analysis and design of nonlinear control systems.
2. The user interfaces of all three tools are steadily progressing away from purely textual specifications of control system design algorithms towards integrated graphical software specification environments. Over the years, the focus of CACSD has matured from a level of subprograms, to individual application programs, to design languages, and finally to integrated design environments. At the same time, the name of the discipline has changed from formerly CACSD (Computer–Aided Control System Design) to newly CACE (Computer–Aided Control Engineering), reflecting the aforementioned shift in focus.
3. The trend goes towards incorporating the object–oriented programming paradigm and world view [5] into CACE environments. XMath, the successor of Matrix$_x$, offers "data objects," which, in terms of traditional software engineering, are simply

strongly typed data structures that enable a more convenient operator overloading capability than that offered in earlier CACSD tools, a need that had long been recognized and had e.g. been promoted in [9]. Unfortunately, this concept has not systematically been carried over to the graphical interface as well. All three graphical environments are based on *block diagrams* rather than the more versatile *object diagrams* proposed in [6,7].

2. HIGH–AUTONOMY SYSTEMS

According to NASA [29], *automation* is defined as "the ability to carry out a pre–designated function or series of actions after being initiated by an external stimulus without the necessity of further human intervention." In contrast, *autonomy* is defined as "the ability to function as an independent unit or element over an extended period of time performing a variety of actions necessary to achieve pre–designated objectives while responding to stimuli produced by integrally contained sensors."

Automation of a process is usually viewed as designing a (feedback) controller that reduces the plant sensitivity to parameter variations and/or the influence of disturbances. Parameter variations may be due to minor variations in the manufacturing process that is used to generate the plant, due to thermal effects, or to the influence of external environmental variations. Mostly, these variations are minor (a few percent). Feedback control architectures have helped to make plant operation more robust (less sensitive) to such types of variations in comparison with open–loop command architectures.

Control theory was developed since it was recognized that there exist patterns between different types of systems that extend beyond a single application or application area. PID–controllers can be used to improve the behavior of large classes of different systems to large classes of different stimuli.

Modern control architectures, particularly the $\mathcal{H}^{\infty}$–control schemes, have focused on making controllers even more robust by enabling them to adequately respond to even larger classes of stimuli, and perform correctly with even less complete plant information [15].

However until now, control engineers hardly ever concerned themselves with abnormal situations such as failures that occur within a subsystem of the plant to be controlled or –even worse– within the controller itself. Reliability of a controlled system over long periods of time is rarely mentioned among the items on the desired performance parameter list of control engineers. They are concerned with such properties as stability, steady–state accuracy, percent overshoot, and settling time; not failure rate, down time, or repair activities.

Such factors must be considered, however, for high–autonomy system operation. They add a new dimension of complexity to the overall system design. A high–autonomy system contains at least one additional hierarchy layer about the conventional control architecture. The overall architecture usually contains several alternative conventional controllers. It knows which of them to choose at any particular moment and when to switch between different controllers. It contains a task planning module that computes the set points for active controllers, and sequences the tasks to be executed. It reasons about both plant and controller integrity, detects faults (symptoms) as they occur, localizes faults within the system (discovers failures), and thinks about means of recovery from such faults (initiates repair activities). It furthermore collects statistics on symptoms, failures, and successful (as well as unsuccessful) repair activities, i.e., it learns from past experience.

In the past, CACSD researchers considered the implementation of the controllers that are being designed by the CACSD tools a fairly straightforward task. Therefore, most CACSD tools output the design specifications either in the form of a state feedback vector or matrix, or maybe a transfer function. In those cases, the implementation of the controller once designed is indeed a fairly simple task that can be easily accomplished using either analog or digital circuitry, or possibly a microprocessor. Only in very special application areas, such as CNC machines or industrial robots, have the CACSD tools directly been merged with the equipment that implements the designed controllers.

In the era of intelligent control, this situation will have to change. Fault–tolerant intelligent controllers for high–autonomy long–term operation of complex plants are so involved that issues of controller implementation can no longer be considered a trivial undertaking by any standards. Consequently, CACE environments will have to be enhanced in yet another direction: not only the user surface has to evolve into an object–oriented interactive graphical design environment, but also the machine interface must be developed into a highly flexible and fairly complex surface able to either transmit commands or control signals directly to the plant and receive telemetry information back for analysis and/or display on the screen, or alternatively synthesize a control program that can be automatically downloaded into another computer for control execution, and that in itself may contain an elaborate user surface and an involved machine interface.

3. SUPERVISORY CONTROL OF REMOTE EQUIPMENT

Teleoperation of equipment in remote locations, such as the moon or deep–sea environments, poses particular problems because of the communication delays between the plant and the remote observer.

Direct man–in–the–loop control is out of the question since the communication delays, which in that situation are part of the feedback control loop, would either render the control system unstable or at least would severely limit the control bandwidth.

High–autonomy control, on the other hand, is often not feasible yet due to weight, volume, and energy restrictions within the umbrella of the overall mission payload. The necessary number–crunching power may not be available at the plant site. This is particularly true for space missions.

A feasible compromise is attained by means of the supervisory control paradigm proposed by Sheridan [34]. This architecture is shown in Fig.1.

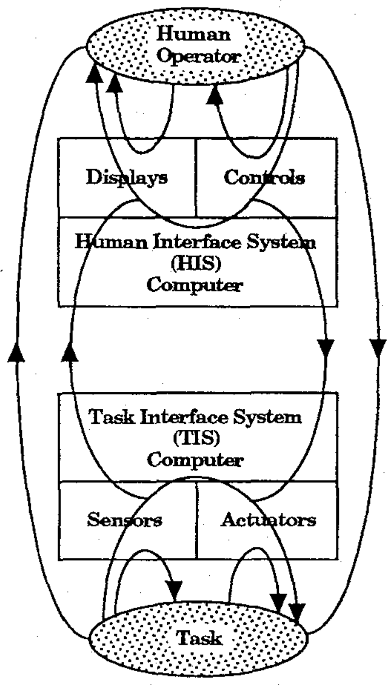

Human Operator
Displays
Controls
Human Interface System (HIS) Computer
Task Interface System (TIS) Computer
Sensors
Actuators
Task

Figure 1. Supervisory control architecture.

In the supervisory control scenario, the "local" plant controllers possess limited intelligence and are equipped to correctly function over a limited time horizon only. Most of the intelligence resides on the "remote" (i.e., observer) site, where it is implemented partly by more intelligent control algorithms (feasible since the number–crunching power at the observer site is practically unlimited), and partly by direct man–in–the–loop interaction. Since local control takes care of the innermost (least intelligent, but fastest) control loops, the communication delays are not critical as long as the local controllers are able to survive longer than the communication delay on their own, i.e., the communication delay dictates the minimum required time horizon of the local controllers.

Software has been developed to support supervisory control of equipment in remote locations. OASIS (Operations And Science Instrument Support) [30] is a software developed for that purpose. Figure 2 explains the layered OASIS software architecture.

Layered architectures to control complexity issues have been in existence for a long time. The well–known ISO Open Systems Interconnection (OSI) standard for layered communication protocols [23] is one example. More recently, the European Computer Manufacturer's Association (ECMA) proposed a similar layered architecture for Computer–Aided Software Engineering (CASE) environments [16].

space applications [19], and remote control of a prototype of an oxygen production plant for planet Mars [10]. However, OASIS does not provide for capabilities related to the *design* of supervisory control algorithms. This could become the task of the next higher level in a layered CACE architecture. Layered CACE architectures have been proposed in the past [3], but it makes little sense to start from scratch. It is much more reasonable to build upon what is already available, namely the OASIS supervisory control environment.

4. INTELLIGENT CONTROL ARCHITECTURE

Conventional control has focused in the past on the control of fully operational plants only. In the context of high–autonomy operation of process plants, conventional control concepts are clearly insufficient.

Fault–tolerant control adds to the system the capability to recognize anomalies in plant behavior, isolate these anomalies within the system, propagate symptoms to failure descriptions, and finally propagate deduced failures to descriptions of corrective actions [28]. However, fault–tolerant control adds to the overall controller complexity, thereby enhancing the risk of controller failure.

Self–aware control reduces this risk again by adding capabilities to recognize and counter anomalous controller behavior. To this end, each controller must contain a supervisory agent that is able to reason about proper execution of the control strategy, and that is able to issue a *declaration of incompetence* in case a controller is unable to perform its designated task. The supervisory agent is equipped to reason about the integrity of the controller it is responsible for, and about the integrity of the supervisory agents of each controller on the next lower layer in the control hierarchy. A declaration of incompetence can result from three causes: (i) a controller malfunctions due to hardware failure or latent errors in the control software, (ii) the controller is faced with a situation outside its operational umbrella, and (iii) the controller itself or its supervisory agent run out of reasoning time (time–out error).

Cognizant control finally adds a capability to reason about the appropriateness of commands issued by the human operators, explain to the humans in clear text the results of this reasoning effort, why, for instance, an issued command sequence may not be safe to execute. It is responsible for carrying on an *intelligent communication* with the human operators, assisting them both during regular operation and emergency processing with current system status assessment and future system status projection.

Intelligent control comprises the entire control envelope including all four control layers (conventional control, fault–tolerant control, self–aware control, and cognizant control). The *conceptual decomposition* of the intelligent control architecture is shown in Fig.3.

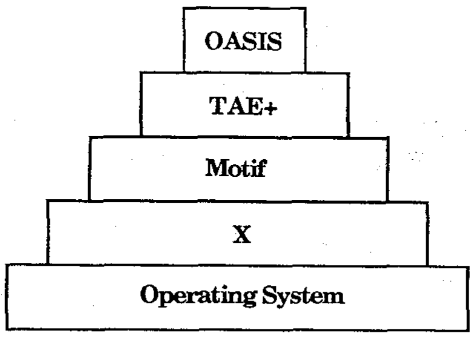


Figure 2. OASIS software architecture.

OASIS realizes a layered software architecture for supervisory control. X [40] is used to manage graphical screens at the bit level. X also offers basic window support, i.e., X provides for the capability of creating multiple virtual screens that reside physically on one real screen and manages the logic behind overlapping windows. X defines a portable standard that allows to address different types of graphical devices in a uniform manner. Motif [21] sits on top of X. Motif concretizes the windows that the user works with. It defines how windows are opened, sized, moved around, and closed. It also defines mechanisms for establishing window tools, such as pull–down menus, but does not define the actions associated with using these tools. TAE+ [37] sits on top of Motif. TAE+ defines the actions associated with window operations. It provides window editors that allow the user to define pointing–device–controlled push buttons and gauges, actions associated with items on pull–down menus, and repositories for data (tables, graphs). Some versions of TAE+ also offer limited resource management capabilities. At the top layer, OASIS [30] controls the data flows. OASIS is responsible for translating push button and pull–down menu actions into command streams (expressed in the CSTOL language [13]), encapsulating commands in command packets (using the CCSDS standard [20]), and transmitting these command packets to the plant site using either DECnet or TCP/IP protocols. OASIS is also responsible for receiving telemetry packets from the plant site, unpacking them, and processing them. Processing of telemetry data usually means to place them in TAE+ data repositories. However, OASIS can also "bridge" incoming data to another computer for further processing, e.g. for the computation of intelligent control actions. Finally, OASIS can be used to compute some control actions on its own, i.e., an incoming data packet may cause a new command stream to be issued.

OASIS goes a long way towards providing a vehicle for the *realization* (i.e., implementation) of supervisory control architectures in a portable (i.e., both platform– and application–independent) fashion, and this task in itself is far from trivial. Recent applications of OASIS include remote control of a simulation model of a space telescope [25], remote control of a prototype of a robotic fluid handling laboratory for

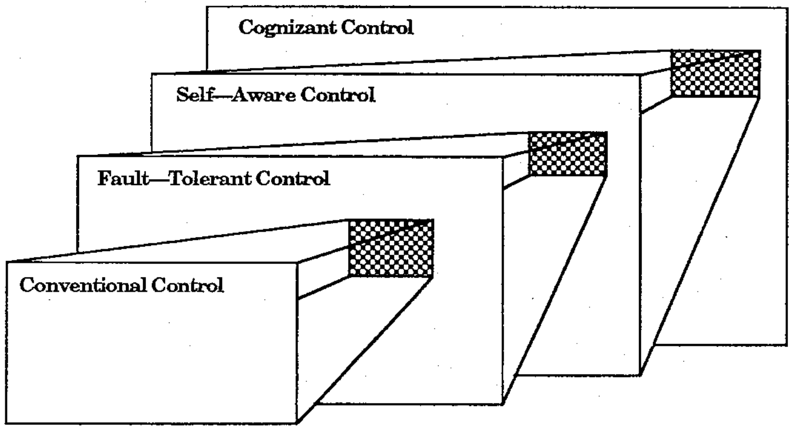


Figure 3. Conceptual control decomposition.

The conventional controller may contain a model of the plant, e.g. in order to solve the inverse dynamics problem. This can be a differential equation model, a neural network model, a fuzzy logic model, or any among a variety of other model types. It is even feasible to use a combination of several such models together with a model selector as suggested in [4]. The fault–tolerant controller may require another model of the plant in order to be able to detect anomalous plant behavior (by comparing the measured plant output with the expected model output when driven by the same control signal). The self–aware controller needs a model of the controller itself probably together with a model of the plant to detect, isolate, and analyze faults in the control architecture. Finally, the cognizant controller needs a model of the control environment. It also needs yet another model of the control architecture. This model must be able to perform a predictive simulation of the control architecture, and is used to generate future system status projections for the consumption of the human operators and possibly the higher–level controllers. This model can e.g. be used to assess the effects of a planned control action before it is actually being implemented on the real plant.

4.1. Fault–Tolerant Control

Fault–tolerant control can be realized using signal envelopes. Signal envelopes can be implemented in the control architecture by means of either state–windows or time–windows [39). Time–windows are best suited to accompany event–based controllers [41]. Violation of state–windows requires watchdog monitors [10,24] for their detection. In the interest of a maximized robustness of the fault–tolerant control architecture, both types of failure detection mechanisms should be implemented in parallel. Watchdog monitors can also be used to periodically test all actuators and sensors to see whether they are still responding in accordance with their designed functionality.

The fault–tolerant control plane further implements a set of fault diagnosers [33] that are able to reason about the cause of an anomaly once detected. Fault diagnosis involves relating symptoms to hypothesized failures. This problem has been studied in

[24,28].

The fault–tolerant control plane also implements a set of fault recovery algorithms. These algorithms usually require replanning, and have been studied by [11,41]. However, many fundamental questions have not been properly addressed yet, e.g., what happens if a new fault occurs during replanning.

4.2 Self–Aware Control

Self–aware control is based on controller models (redundancy of control signal computation) and feedback from the plant. Smart plant actuators acknowledge receipt of (discrete) control signals, and smart plant sensors acknowledge accomplishment of the commanded tasks. Time–windows can be used to verify that requested actions have been accomplished in accordance with the designed specifications.

Other self–aware control concepts include:

(1) *Robustness:* Each procedure must have an umbrella to recognize its own limitations. As a procedure reaches that limit, it declares itself incompetent, and passes the responsibility to the next higher level in the control hierarchy. Thus, in this architecture, robustness is a bottom–up process initiated at the lowest levels of the control structure by the declaration of incompetence.

(2) *Alarming:* Incompetence declaration is linked to alarm levels. As long as all active procedures are competent, the plant is in a "green" state. As soon as one or several low–level controllers declare themselves incompetent, the plant proceeds to level "yellow" with prescribed actions taking place at that point. If the next higher–level controllers declare themselves equally incompetent, the plant proceeds to level "orange," etc.

(3) *Alarm Filtering:* As a plant or controller perturbation propagates upwards through the control structure, the characterization of the perturbations progress from a local, sub–system specific, characterization to a broader system–based characterization. High level information concerning the fault is developed automatically as the control system evaluates its own performance.

4.3 Cognizant Control

The most important principle in the context of cognizant control is the capability to perform a predictive simulation of the overall control architecture. In this way, commands issued by the human operators (or by higher–level controllers for that matter) can be simulated ahead (using a model of limited validity to make it execute fast). If these commands don't lead to simulated anomalies within a reasonable amount of (simulated and real) time, it can be assumed that the action is safe, and the action can be forwarded to the real plant. However, if a simulated anomaly is detected, the fault diagnosers can be activated to reason about the cause of the simulated anomaly. Simulated alarm filtering will propagate the reasoning process back to the operator who then is given the opportunity to either confirm or cancel the requested command.

The same feature can also be used for an entirely different purpose. Operators can issue a command to the simulator only, i.e., they can inform the control system that they don't wish this command to be actually implemented. This facility can be used both for operator training on the real system and for troubleshooting the system while it is in full operation.

Finally, it is extremely important that the operators develop trust in the control architecture [31]. They will never trust the controllers if they don't understand what the controllers are doing when and why. Therefore, it is important that the cognizant controllers contain a "model" of the human operators that they use to provide the operators with the kind of information that they would expect in order to be assured that the control architecture is still working reliably [35]. To give an example: it may be necessary for the higher–level controllers to test the lower–level controllers from time to time by going into a testing mode. They must never do so without informing the operators of what is on their "mind." There is nothing more disconcerting for an operator than a control system that, without any obvious cause, suddenly becomes "active" and changes its behavior. On a similar note, it is important that the high–level controllers keep the operators "entertained," i.e., provide them regularly with summary information since otherwise the operators might get nervous and suspect that the control system has stopped to operate. Yet, the high–level controllers should never "volunteer" more information than human operators can comfortably digest at any one time. Of course, the operators can actively request information whenever they want and to whatever extent they deem suitable.

5. KNOWLEDGE REPRESENTATION AND PROCESSING

While the previous section introduced a conceptual decomposition of the intelligent control architecture, it did not outline how the control knowledge is actually being processed. For this purpose, a *granular decomposition* of the control architecture is presented next.

Central to the realization of the objectives of intelligent control is an organized mechanism for knowledge representation and processing. Such a mechanism should include appropriate transformations between diverse knowledge types, such as numeric, symbolic, linguistic, etc. residing at different parts of the generic intelligent control architecture in order to provide proper interfacing in the execution of the overall control task. While traditional control design procedures have paid a greater degree of attention to purely numerical knowledge processing, relatively little effort has been spent on the processing of other forms of knowledge.

A granular hierarchy in the overall processing of knowledge for the control of a complex dynamical system is shown in Fig.4. This figure also identifies the principal features of the represented knowledge at the various stages and the possible tools for processing of this knowledge.

The granular decomposition is responsible for the translation of knowledge between different levels of abstraction (granularity). This translation works both ways. High–level commands issued by human operators need to be translated down to actual control actions at the conventional control level, and measurement data need to be processed (filtered) for consumption by the higher–level controllers and ultimately the human operators. The data filtering process relates to both presentation of system status during normal plant operation, and alarm filtering during emergency processing.

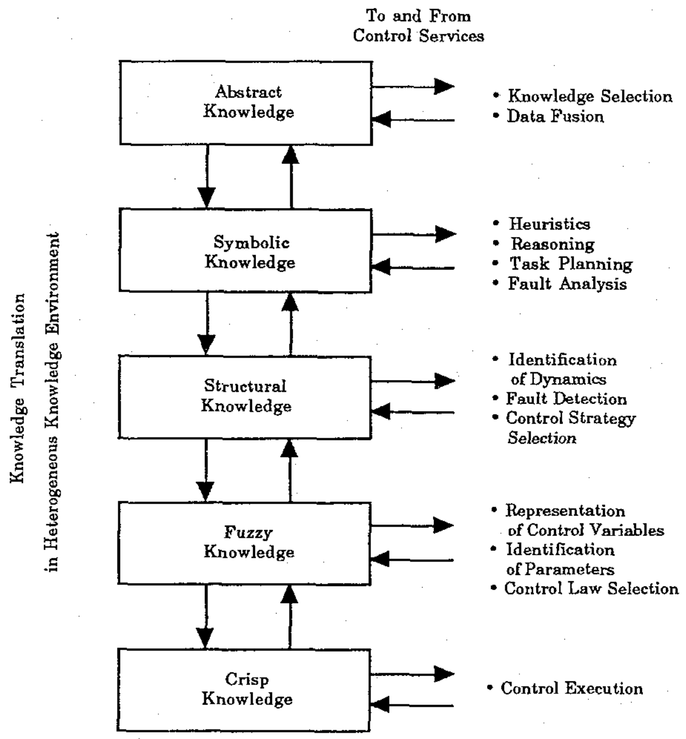


Figure 4. Granular control decomposition.

6. FUNCTIONAL CONTROL HIERARCHY

While the previous section discussed the translation of knowledge for the intelligent control architecture, it did not outline how the control hierarchy is actually being realized. For this purpose, a *functional decomposition* of the control architecture is presented next.

Figure 5 shows the functional decomposition of the intelligent hierarchical control architecture. While the granular decomposition is responsible for the translation of knowledge, the functional decomposition accomplishes the decomposition of actual control tasks. The *task planner* is responsible for translating high–level managerial commands into sequences of plant commands [12]. It is responsible for resource management, i.e., ensures that the resources needed to perform a particular task are available at the time when the task is executed. The task planner has a considerable degree of freedom in determining which tasks are executed when and in what sequence. The *task scheduler* decomposes task requests from the task planner into individual steps. The difference between task planning and task scheduling is that the sequence of steps within a task is always fixed, i.e., a single task can be viewed as a macro–step. The *command interpreter* puts individual steps into action. It is responsible for preparing the control architecture for execution of the next step. It selects the proper controllers from the controller library, downloads them into the controllers, and kicks off the actual implementation. The *command executor* finally represents the actual controller that carries out the commanded action.

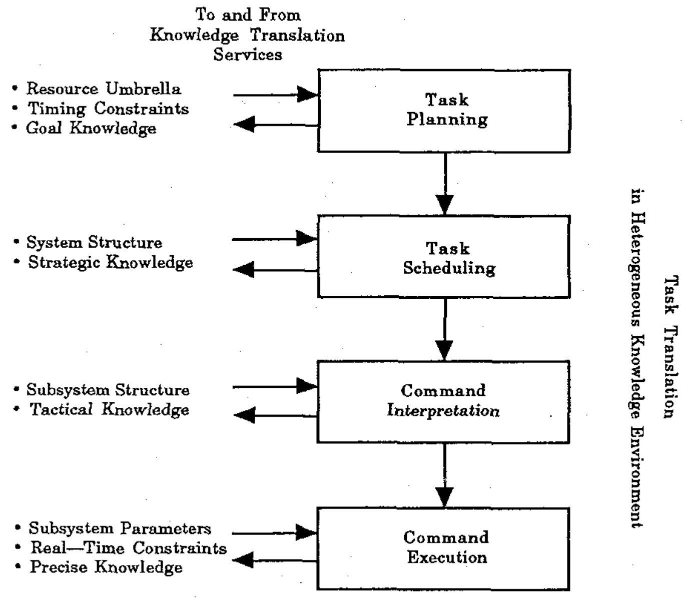


Figure 5. Functional control decomposition.

Numerous such architectures have been proposed in the past, most prominently the NASREM robotics architecture [2]. It should be noticed that no direct correspondence exists between the functional and granular decompositions. The granular decomposition corresponds loosely to the world model column in the NASREM architecture, but it is misleading to represent both decompositions on the same figure. The granular decomposition simply provides the knowledge that is being used by the control services in whichever way needed.

The functional decomposition explains how complex plants can be controlled in a high–autonomy mode of operation. It does not discuss yet the nature of the actual controller itself. The command executor can be a simple PID controller in some cases, an optimal or adaptive controller in others, or something more exotic such as an event–based controller [41] operating on time–windows [39].

In particular, the command executor can itself be realized as a hierarchical control structure. Such a structure is shown in Fig.6. This architecture can be regarded as a generalization of some well–known structures such as the action–critic control structure and the master–slave formulations that are popular in neural network–based processing. In the structure shown in Fig.6, Level V represents the processing of higher–level functions (the reasoner), while Level L encompasses the remainder of the functions leading to the exact determination of the control variables to be input into the system to realize the desired objectives.

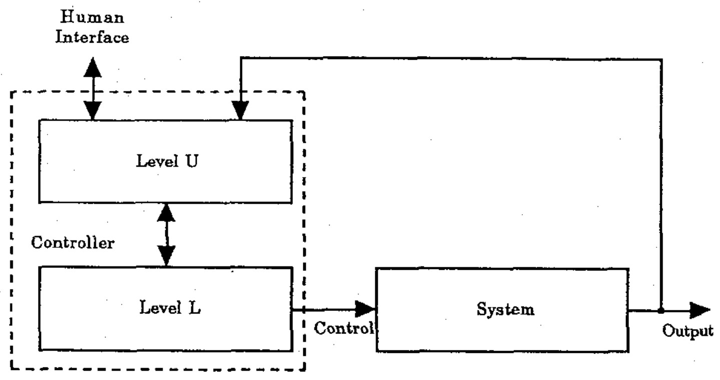


Figure 6. Operational control decomposition.

It must be emphasized that each of the two levels shown could be decomposed into a number of sublevels appropriate for a specific purpose. Some characteristics that guide in the determination of the number of sublevels are the types of knowledge used in the processing, the control objectives, the complexity of the mathematical representation of the system dynamics, and the extent to which human operators are allowed to intervene directly in the system operation.

It must also be stressed that complex plants contain many control variables. Obviously, each control variable can be computed by a separate controller realized in a *distributed control architecture*. Thus, all of the previously discussed hierarchies can be duplicated as needed to support the distributed controllers used to implement individual control loops for different control variables.

7. CONTROL OF A MARTIAN OXYGEN PRODUCTION PLANT

7.1. Plant Description

The University of Arizona / NASA Space Engineering Research Center for Utilization of Local Planetary Resources (NASA/UA SERC/CULPR) is investigating means to generate oxygen from lunar and/or asteroidal rocks as well as from the Martian atmosphere. In particular, the Martian oxygen production plant has progressed beyond mere simulation to the status of rapid prototyping. Figure 7 shows diagrammatically the Martian oxygen production prototype.

The Martian atmosphere (90% CO_2 at a pressure of 6 mbar and at a temperature of 200 K) is condensed (and thereby heated) in a compressor to a pressure of 1 atm. It is then heated further to a temperature of roughly 900 K. At that temperature, carbon dioxide (CO_2) decomposes through thermal dissociation into carbon monoxide (CO) and oxygen (O_2). Unfortunately, these two gases have similar molecular weights and are therefore difficult to separate. The heart of the system is an array of zirconia tubes. These tubes separate the two gases in an electrocatalytic reaction. The oxygen is liquefied for storage, whereas the components of the CO/CO_2 gas mixture are separated in a membrane separator. The CO_2 is then rerouted, whereas the CO is further processed by a Sabatier process to generate methane (not shown on Fig.7).

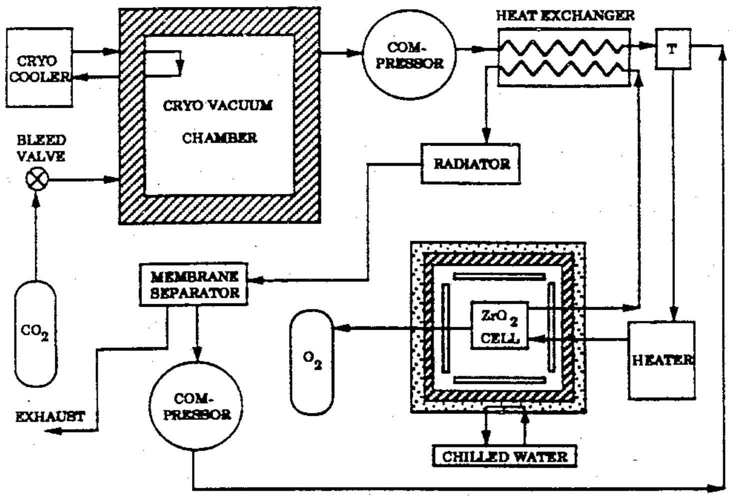


Figure 7: Martian oxygen production plant.

In space exploration, large amounts of oxygen are used for propulsion. In any chemically propelled spacecraft, a *fuel* reacts with an *oxidizer*. Together they constitute the *propellant*. The simplest chemical reaction that can be used for propulsion is:

$$2H_2 + O_2 \rightarrow 2H_2O \tag{1}$$

During the reaction, energy is freed that can be used to propel the spacecraft. In this propellant, the fuel (liquid hydrogen) makes up only 11% of the weight whereas the oxidizer (liquid oxygen) occupies 89%. While many different combinations of fuels and oxidizers can be employed for propulsion, oxygen is the most commonly used oxidizer. In all such reactions, the oxidizer is considerably more heavy than the fuel. For these reasons, it is economically interesting to generate oxygen on other planets.

The 16–tube breadboard generates roughly 1 kg of oxygen per day. The methane that is generated by the Sabatier process can be used as fuel. The current testbed could lead to a flight–rated plant before the turn of the century to be launched to planet Mars by NASA in an unmanned mission. A later manned mission could then use the oxygen that would meanwhile have been produced as oxidizer for the return flight to Earth. In this way, the manned mission could arrive on planet Mars with empty tanks, and the propellant for the return flight would not have to be lifted out of the gravity well of planet Earth. A manned mission to Mars could take place as early as 2014.

7.2. Command and Control Architecture

Figure 8 shows the overall command and control architecture.

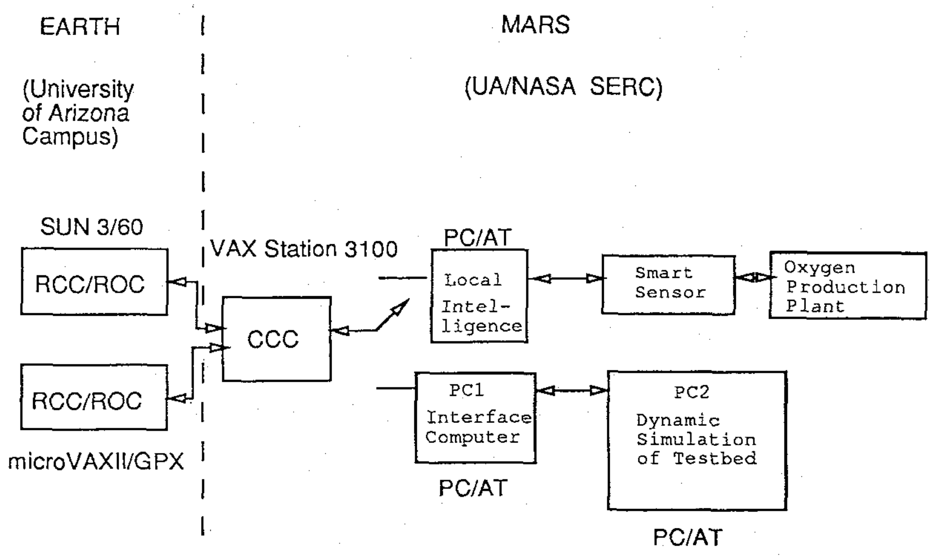


Figure 8: Command and control architecture.

Notice the usage of the terms "remote" and "local." In the terminology employed here, "local" stands for the plant site, whereas "remote" denotes the human commander. This nomenclature comes naturally to control engineers who view their plant as central. Psychologists have a tendency to view the human commander as central, and therefore call that location "local" and the plant site "remote."

At the remote site, several remote observers (ROCs) can share in monitoring an ongoing experiment. Every one of the ROCs runs the OASIS software. One of the remote OASIS workstations is the remote commander (RCC), who actually commands and controls the experiment. Only one commander is allowed to issue *intrusive* commands to the plant at any one time. However, one or several other remote sites can obtain telemetry packages from the plant. They are also allowed to send *non-intrusive* commands to the plant. A key assignment mechanism is used to determine who, among the remote participants, is the commander. The rationale behind this decision is the following: several human experts may be located at several different sites on Earth. If a particular type of expert knowledge is required during a particular phase of the operation, the most knowledgeable person will be given the command key, while all other participants become passive observers. They can communicate with the momentary commander through "open microphones." If, during a later phase of the operation, another participant becomes more knowledgeable, the command key can be passed on to that individual, and the former commander becomes now one of the observers.

At the local site, a command and communication center (CCC) is responsible for the communication with the RCC and the ROCs. The CCC is also the key manager. In the future, a second CCC will be added at the remote site, thus, the RCC and ROCs will communicate only with the Earth–bound CCC, whereas the processing plants will communicate only with the Mars–bound CCC. The long–distance communication will be limited to a communication between the two CCCs.

The CCC decodes command packets, and passes them on to the appropriate local controlling computer (LCC) for further processing. The CCC can be used to manage several LCCs simultaneously controlling several different pieces of equipment. The LCCs are responsible for the processing of local intelligence. They monitor their respective plant within the umbrella of local control. A limited set of intelligent decisions can be made by the LCCs directly without conferring with their RCC.

However, the LCC is not responsible for managing the sensors and actuators, and for implementing the actual low–level control. This task is reserved for the "smart sensor." The smart sensor offers fast sensor monitoring and actuation, A/D and D/A conversion, time–multiplexing, as well as a limited amount of programmable logic control. However, since the memory available for storing control programs is very limited, these programs must be frequently exchanged. Therefore, the LCC assumes the role of the *command scheduler*, whereas the smart sensor assumes that of the *command executor*.

7.3. Task Planning and Scheduling

The task planner used in the oxygen production prototype is fairly rudimentary. An elaborate task planner was not required. Figure 9 shows how the task planner manages the resource and time umbrellas for four different control loops during the start–up sequence, the steady–state operation, and the shut–down procedure. The four control loops are concerned with voltage control, temperature control, flow rate control, and control of a gas chromatograph.

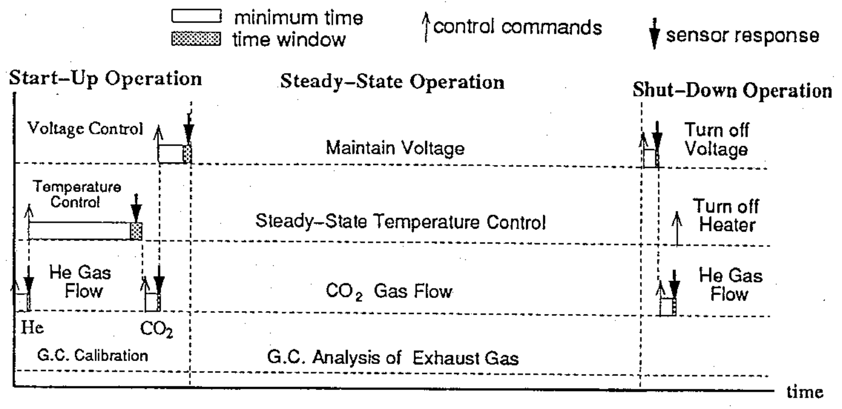
The first control loop is concerned with the voltage applied across the wall of the zirconia cell. During the separation of oxygen and carbon monoxide, a voltage of 2 V DC must be applied to the zirconia cell for the electrocatalytic reaction to take place. In the presence of zirconia, the hot oxygen gas is ionized by borrowing four electrons from the cathode located at the inside wall:

$$O_2 + 4e^- \rightarrow 2O^{2-} \tag{2}$$

The oxygen ions are sufficiently small to migrate through the porous zirconia wall. Upon arrival at the other side, they shed the electrons to the anode:

$$2O^{2-} \rightarrow O_2 + 4e^- \tag{3}$$

and free oxygen is released.



System State Variable Table

temperature		25
	upper bound	30
	lower bound	20
flow rate		0.07
	upper bound	0.08
	lower bound	0.06
voltage		0
	upper bound	0
	lower bound	0
input gas type		He

Start-Up Operation

System State Variable Table

temperature		900
	upper bound	915
	lower bound	885
flow rate		0.07
	upper bound	0.08
	lower bound	0.06
voltage		2.00
	upper bound	2.05
	lower bound	1.95
input gas type		CO_2

Steady-State Operation

System State Variable Table

temperature		25
	upper bound	30
	lower bound	20
flow rate		0.07
	upper bound	0.08
	lower bound	0.06
voltage		0
	upper bound	0
	lower bound	0
input gas type		He

Shut-down Operation

Figure 9: Task planning for Martian plant.

The oxygen production rate is very sensitive to the applied voltage. Below 0.6 V (Nernst voltage), no oxygen is produced at all. For higher voltages, the oxygen produced is linear in the applied voltage. However, at 2.3 V, the zirconia cell is permanently destroyed. However, there is also a relation between voltage and temperature. No voltage should be applied to the cell before steady-state temperature and gas flow rate have been reached, and the voltage should be switched off at the beginning of the shut-down sequence.

Temperature control is also essential. A temperature of at least 850 K is necessary for thermal dissociation of the carbon dioxide to take place. However, at temperatures between 1000 K and 1200 K, the zirconia cell is permanently damaged (the critical temperature depends on the material used for electrodes and seals). There also exists a relation between temperature control and gas flow. Obviously, the proper gas flow rate must be established before heating can take place. However, around 600 K, there is a risk of carbon deposition along the walls of the zirconia cell. For that reason, the cell is being purged with helium gas during start-up and shut-down. The helium gas is only replaced by carbon dioxide gas after the zirconia cell has reached its operational temperature of 900 K.

The third control loop is concerned with maintaining the appropriate gas flow rate and with operating the electronically controlled valve that switches between the helium and carbon dioxide gas.

The fourth control loop operates the gas chromatograph that is used to analyze the various gases during the experiment. During start–up, the gas chromatograph needs to be calibrated, and during steady–state, various gas samples can be routed through the gas chromatograph by appropriately setting several valves.

Figure 10 shows the physical configuration of a single–cell breadboard used during the initial phase of this project.

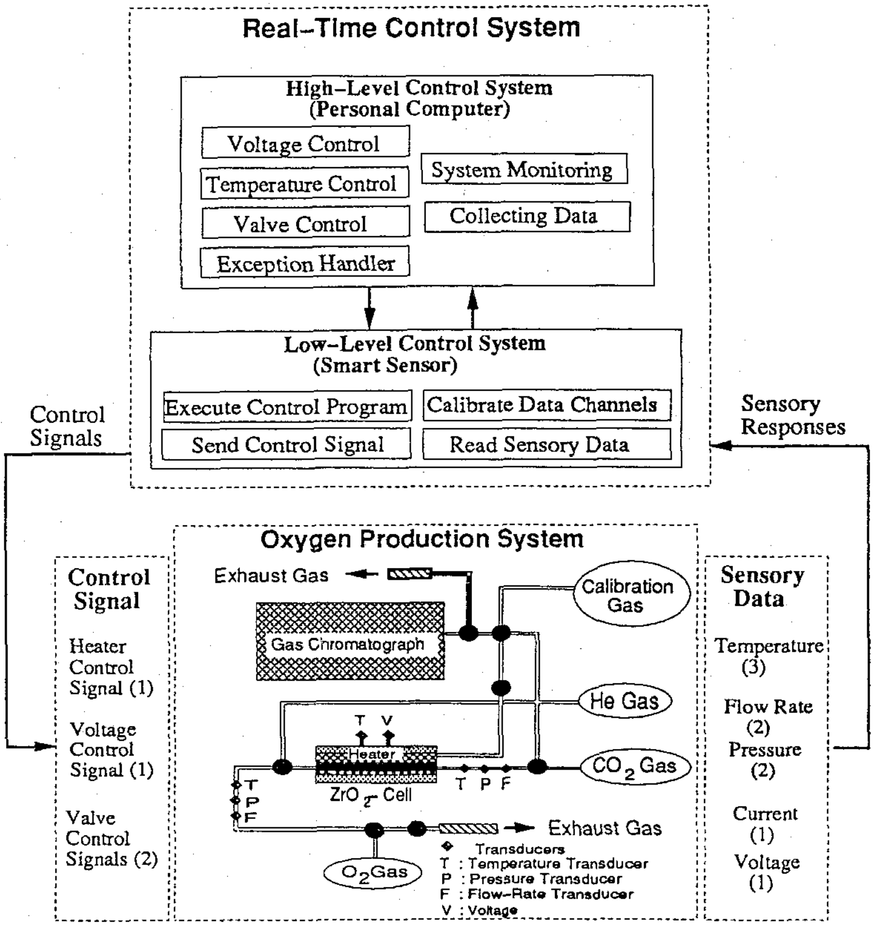


Figure 10: Configuration of single–cell breadboard.

While the task planner is currently still very simple, this part of the program will need to be drastically expanded before the full–scale plant can be launched. Task planning is not only responsible for setting the pace for normal operation, but also for recovering the plant after a fault has occurred. The Martian task planner must be able to deal with sand storms, contaminated membranes and tubes, leakage of seals, and tripped circuit breakers, to mention just a few of the types of failures that *are expected* to occur during long–term operation of this high–autonomy control system.

7.4. Command Scheduling

The command scheduler is an independent agent that resides in the local controlling computer. It accepts the next command to be executed from the task planner and prepares the control architecture for its execution. The command scheduler selects the appropriate low–level control algorithm from a set of precoded algorithms, downloads the selected controller into the smart sensor, and initiates the control activity.

The command scheduler is also responsible for receiving and processing sensory events that signal task completion. It is responsible for maintaining the appropriate time–window information that allows it to judge success or failure of task execution. Consequently, the command scheduler is responsible for fault detection during transient operational phases, such as the start–up and shut–down phases that were previously discussed. In case of a failure (the sensory event has arrived *too–early* or *too–late*), it triggers the fault diagnoser, which starts to reason about the nature of the observed fault so as to relate the observed symptom to the failure that caused it.

Time windows are intimately linked to event–based control logic. They ensure early detection of faults during transient operational phases, and thereby provide the high–autonomy system with sufficient *reliability* to allow the system to operate adequately over an extended period of time. Figure 11 explains the time–window mechanism.

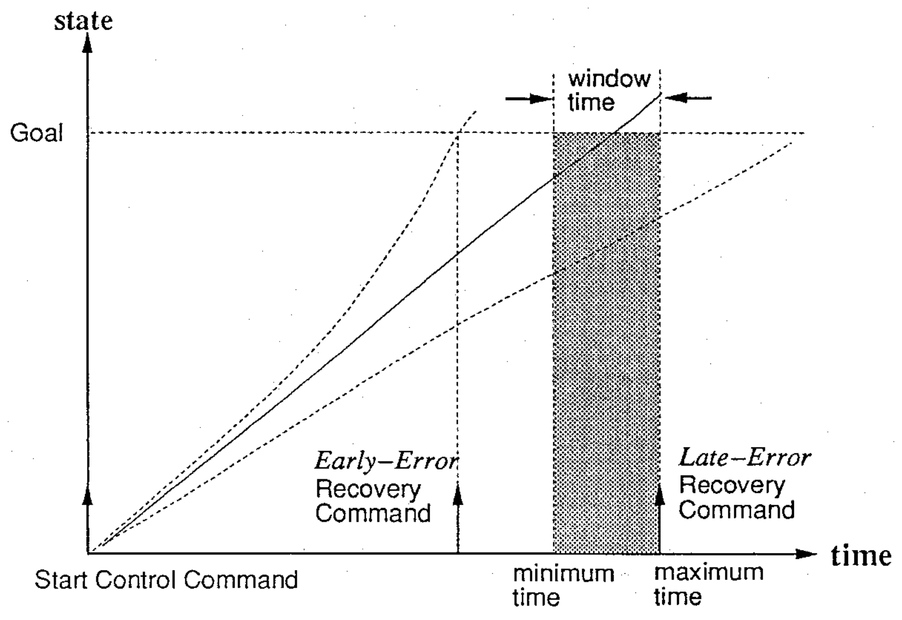


Figure 11: Time–window mechanism.

Each command issued by the command scheduler is accompanied by an expectation as to how much time the commanded task will require for its completion, i.e., before the goal state is reached. Due to plant parameter uncertainties and external disturbances, the time cannot be estimated precisely. Instead, the uncertainty is captured in a so–called time window. A sensor should record the fact that the next goal state has been reached, and should report the time of task completion to the command scheduler.

If the sensory event arrives *too–early* or *too–late*, a fault diagnoser is activated. The objective of the diagnoser is to relate the observed symptom back to its cause, the failure. It then calls upon the failure recovery agent to come up with a new command sequence that will put the high–autonomy system back on track. Once the failure recovery agent has decided what to do, it calls upon the task planner to work out a new command sequence that implements the suggested recovery action.

The time–window mechanism has been described in detail in [39]. The time windows associated with the control experiment are shown on Fig.9.

7.5. Command Execution

The actual low–level control is implemented in a microcontroller called a smart sensor. Once the command scheduler has downloaded a control program into the smart sensor, the low–level controller is activated. This can be a classical controller of any vintage. Figure 12 shows the two–level local control architecture employed in this testbed.

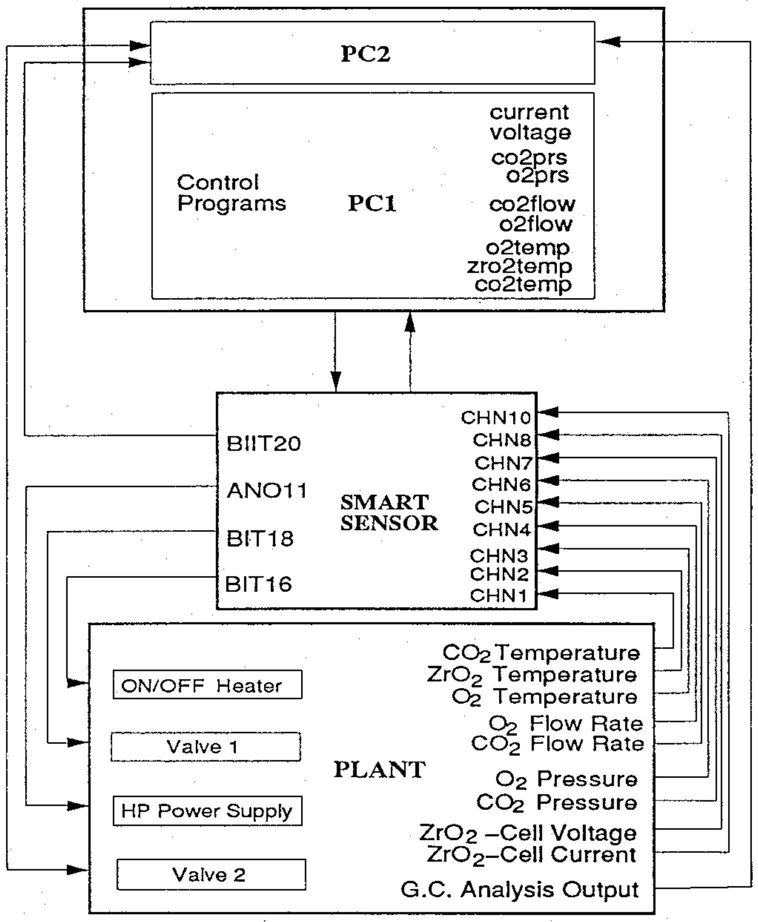


Figure 12: Two–level local control architecture.

The smart sensor accepts sensory information from the plant. Sensory signals are then time–multiplexed before they are forwarded to the local controlling computer (a PC). The smart sensor also receives multiplexed control signals from the PC, unpacks them, and sends them out through its actuator channels to the plant. In this mode, the smart sensor does not perform any control at all, but only serves as an interface between the plant and the local controller. However, the smart sensor can be programmed to either preprocess the sensory signals before sending them to the PC (sensor fusion), or post-process the actuator signals before sending them to the plant (actuator expansion), or compute actuator signals directly from the sensory information received. In the latter case, the smart sensor acts as a microcontroller, and that is the mode, in which the testbed is operated. The memory of the smart sensor used here is very limited. Therefore, the microcontrol programs must be short. For that reason, different control programs are downloaded into the smart sensor from the PC during various operational phases. However, using the smart sensor as a microcontroller (programmable logic controller), it is possible to ensure high–speed control, relieve the PC from low–level control activities, and reserve its computing cycles for the higher–level intelligent control decisions.

In the testbed, all control activity was event–based, but there is nothing in the advocated methodology that would dictate such a solution. It should be noticed that the control programs are indeed different during different phases. For example, the voltage is carefully ramped up in the start–up phase by one control program (to avoid overshoot), whereas the voltage control program operates quite differently during steady–state operation.

It is useful to discuss temperature control in a little more detail. During start–up, the heater is simply switched on. When the temperature reaches a value of 890 K, a threshold sensor detects this fact, and sends a sensory event back to the smart sensor that is then forwarded to the PC. The PC compares the time of arrival with the time window specified for (electrically) heating the system. If the completion event arrives *too–early*, something bad must have happened. Since the electrical heater is operating at its maximum power, the completion event should never arrive *too–early*. If this is nevertheless the case, the fault diagnoser concludes that there must be something drastically wrong in the system, and the recovery unit decides that this fault is not reparable and instructs the task planner to design a graceful shut–down sequence starting from the current state. If the completion event arrives *too–late* (or rather, it doesn't arrive in time), the fault diagnoser concludes that either the power supply or the heater is not functioning properly. In this case, the recovery agent switches both the power supply and the heater off and back on, sends another command to the power supply to reset its voltage, and instructs the task planner to try again with a new time window. If the second attempt fails also with a *too–late* indication, the recovery unit concludes that the situation is hopeless and instructs the task planner to design a graceful shut–down.

During steady–state operation, temperature control is governed by three rules:

1. **Rule_1: Heating** If the current temperature, *Temp*, is below 890 K, turn the heater on and enable Rule_3.
2. **Rule_2: Cooling** If the current temperature, *Temp*, is above 915 K, turn the heater off and disable Rule_3.

3. **Rule_3: Approach** If the current temperature, $Temp$, is between 890 K and 898 K, turn the heater on, estimate the expected remaining heating time τ_{heat}, which is proportional to the difference of the goal temperature, 900 K, and the current temperature, $Temp$, and schedule a turn–heater–off event to occur τ_{heat} time units into the future.

The rationale behind this event–based control logic is simple: since the temperature sensor is not directly attached to the heat source, a certain overshoot behavior is possible. To prevent the temperature from overshooting, Rule_3 is used. This rule functions similarly to a corrective maneuver of a spacecraft: the amount of necessary correction is computed beforehand and then the vernier engine is fired for a precomputed period of time. To avoid problems with local heat stow, the goal temperature is always approached from below, i.e., Rule_3 is disabled during cooling periods until the temperature has fallen safely below the goal temperature everywhere.

7.6. Watchdog Monitors

No time windows are shown on Fig.9 during steady–state operation of the oxygen production plant. This is understandable since, during *normal* steady–state operation, the same control algorithms are constantly active, and neither the task planner nor the command scheduler have anything to do.

Watchdog monitors are independent intelligent agents that monitor the high–autonomy system during steady–state operation. They have knowledge of some components of nominal system behavior during steady–state, and compare their expectations with the actually observed behavior. If a significant discrepancy is found, the "disquieted" watchdog alerts a fault diagnoser to come up with an explanation for the observed anomaly.

The watchdog monitor philosophy has been advocated in [24]. In the incident described below, it was one of the watchdogs that finally, and unnecessarily late, got aroused and triggered off the event chain that ultimately led to the restoration of the "distressed" temperature controller.

7.7. Fault Diagnosis

Contrary to the watchdog monitors that are active *daemons* throughout the steady–state operational phase, fault diagnosers are dormant sequential routines. They are activated only after an anomaly has been detected. The purpose of a fault diagnoser is to relate an observed symptom back to the failure most likely to have caused it.

In the testbed, only an extremely simple global rule–based fault diagnoser was employed. Those failures from which the high–autonomy architecture can recover are indicated by clear symptoms, and therefore, no complex fault diagnosers are needed. However, before an actual plant can be launched, it must be ensured that the high–autonomy system can recover from *all* foreseeable sorts of mishap. It will then become essential that the precise nature of any observed failure is well understood before an automated repair activity is initiated. For that purpose, a multi–level hierarchical model–based diagnoser will be needed. Such an architecture was first proposed by [33]. It has meanwhile been elaborated upon by [11].

7.8. Fault Recovery

The findings of the fault diagnoser will invariably be forwarded to a fault recovery agent. It is the task of that agent to decide whether something can be done about the failure or not.

If recovery is possible, it is the job of the recovery agent to compute a new *goal state.* It then provides the task planner with the current state and the desired goal state, and requests that a new command sequence be computed that moves the high–autonomy system from the current state to the desired goal state.

If no recovery is possible, the recovery agent will provide the task planner with the current state only and request computation of a command sequence for graceful shut–down.

7.9. Description of an Incident

The following section uses actual data from a portion of a 100 hour test run to illustrate the operation of the agents described above.

Figure 13 shows the zirconia cell temperature between 43,000 seconds and 107,000 seconds from start of the test. This is a portion of the steady–state operation.

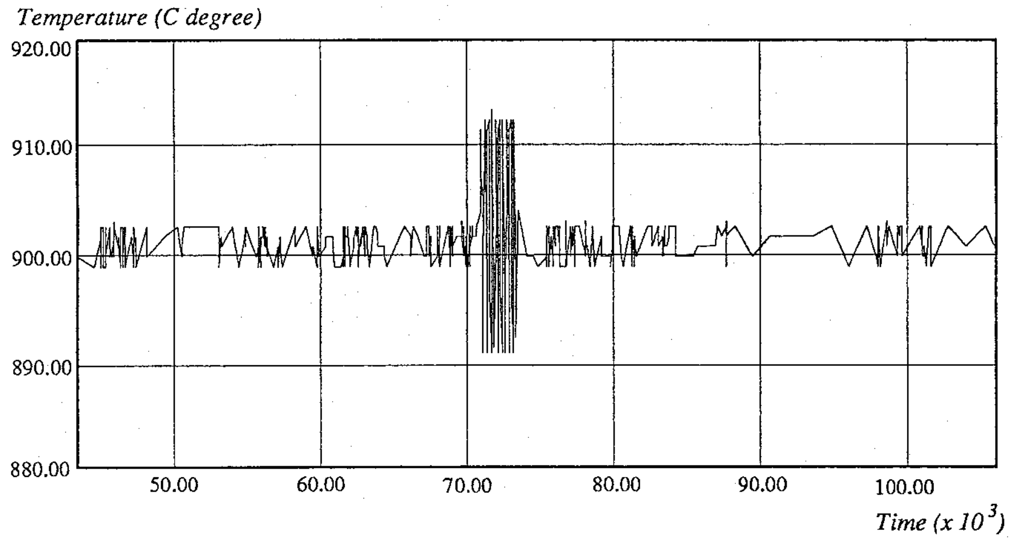


Figure 13: ZrO_2 temperature plotted over time.

It is immediately visible that around 71,000 seconds, something strange happened. The system recovered from the anomaly roughly 2,000 seconds later.

What happened? The anomaly started while a heavy thunderstorm took place. This was during the evening, and no human observer was at the plant site. From other curves, it can be concluded that at that time, there was a short power failure (less than 2 seconds). The backup power supply took over, but there are signs of a temporary power surge. Somehow, this transient upset the smart–sensor–based temperature controller.

It is still not clear exactly what happened. It must be assumed that the power failure stopped the real–time clock of the smart sensor. Thus, when Rule_3 fired, it computed the remaining heating time, but the turn–heater–off event never arrived since the real–time clock wasn't working. Thus, the heater stayed on until Rule_2 fired, which disabled Rule_3. Consequently, the system then cooled itself down until Rule_1 fired, which reenabled Rule_3, which computed a new remaining heating time, etc.

It should have been easy to detect this anomaly immediately. However, the software being used did not require that the value of the real–time clock of the smart sensor be reported back to the PC, and thus, none of the watchdog monitors suspected any trouble. In normal operation, Rule_2 should never be fired. Thus, the first firing of Rule_2 should have sufficed to alert a fault diagnoser — but this case hadn't been foreseen, and the firing of Rule_2 during steady–state operation had not been declared as an anomalous event. Finally, one of the watchdogs got suspicious — unnecessarily late. Since temperature change is normally such a slow phenomenon, that particular watchdog was executed only once every 30 minutes to save computing cycles on the PC. This is a serious drawback of the watchdog monitor philosophy *per se*: since watchdogs are daemons, they must be executed repetitively even if nothing is wrong. If they are executed rarely, they aren't very effective, but if they are executed frequently, they consume lots of computing power. Fortunately, since they are daemons, they run asynchronously and could be installed on separate CPUs, even though that was not done in this testbed.

Fault diagnosers are harmless. They are sequential routines and don't consume any computing cycles unless an anomaly has been observed. In this incident, the fault diagnoser, once invoked, worked beautifully. It concluded that the failure had to be in the smart–sensor program. It did not conclude that the bug was in the real–time clock, but this wasn't necessary. As an analogy: if a computer is down, the repair person will identify the fault only up to the board level and exchange the entire board. The faulty board can then be taken back to the lab where it can be further analyzed down to the chip level. There is no need to perform the second type of fault diagnosis on–line and in real time.

The recovery agent then decided to reload the control program once more into the smart sensor memory. It would probably have sufficed to restart the smart sensor without replacing any programs, but the additional action performed was harmless and didn't consume much time anyway.

8. CRITICAL ANALYSIS

While the previously described testbed is still fairly simple, it contains all elements of the advocated intelligent control architecture. It implements the entire range of the conceptual decomposition. Both fault–tolerant and self–aware control concepts are realized (in a rudimentary fashion) within the local controlling computer. In the described incident, the self–aware component of the control architecture was able to discover, diagnose, and recover from a fault in the controller itself that furthermore *had not been foreseen*. This is a quite spectacular result. Even portions of the cognizant control layer were implemented in the remote commanding computer. Commands issued by the human commander are analysed within the RCC for consistency and compatibility. *Hazard windows* pop up on the screen when the commander tries to command the testbed in a potentially dangerous fashion. Furthermore, it is possible to send a command to the plant simulator rather than to the plant itself to analyze the effects of a proposed action prior to its implementation on the real plant. Also the functional decomposition has been implemented. No clear separation between task planner and task scheduler exists in the current testbed. However, meaningfulness of a separation between command scheduler and command executor has been convincingly shown.

The example demonstrates clearly that, in the context of intelligent control, implementational issues are by no means trivial. Programming of the testbed was slow and painful. Worst of all were issues related to communication protocols among the various inhomogeneous computing platforms involved in the testbed. No CACE tools were used in the design of this testbed ... because none were available. An adequate CACE tool should support the user in the design of the overall intelligent control architecture, it should then be able to simulate the intelligent control system with sufficient realism to judge its acceptability, and it should finally be able to generate code that can be downloaded into the various real–time computers that implement the intelligent control architecture.

These demands are a tall order, and the state–of–the–art of current CACE technology is far from being able to fill it. This is the challenge of the nineties.

9. SUMMARY AND CONCLUSIONS

Multi–Facetted Hierarchical Decompositions were introduced and used as the corner stones of a generic intelligent control architecture. The *conceptual decomposition* captures layers of increased awareness and thereby intelligence; the *granular decomposition* encodes layers of increased knowledge abstraction; the *functional decomposition* describes layers of increased task abstraction; and the *operational decomposition* pertains to layers of reasoning abstraction in the context of the control objective.

The layers of increased awareness relate to *conventional control*, dealing with fully operational plants only; *fault–tolerant control*, able to recognize and compensate for faults in the plant; *self–aware control*, capable of detecting and correcting errors in the controllers themselves; and *cognizant control*, geared to perceive and respond to erroneous behavior of the control environment.

A prototype of a Martian oxygen production plant served as a vehicle to concretize the intelligent control architecture. It was demonstrated by means of this testbed, which implementing all major components of this architecture, that this design enables intelligent controllers to recover from even unforeseen types of failures in an efficient and organized manner.

It was demonstrated that today's CACE tools are insufficient to support the design of intelligent controllers. Hopefully, the next generation of CACE tools will remedy this deficiency.

10. REFERENCES

1. Agathoklis, P., F.E. Cellier, M. Djordjevic, P.O. Grepper, and F.J. Kraus, (1979). "INTOPS, Educational Aspects of Using Computer–Aided Design in Automatic Control," *Proceedings IFAC Symposium on Computer–Aided Design of Control Systems* (M.A. Cuénod, Ed.), Zürich, Switzerland, August 29–31, Pergamon Press, Oxford, U.K., pp. 441–446.
2. Albus, J.S., H.G. McCain, and R. Lumia, (1987). *NASA/NBS Standard Reference Model for Telerobot Control System Architecture (NASREM)*, NBS Technical Note 1235, Robot Systems Division, Center for Manufacturing Engineering, National Technical Information Service, Gaithersburg, Md.
3. Barker, H.A., M. Chen, P.W. Grant, C.P. Jobling, and P. Townsend, (1992). "An Open Architecture for Computer–Assisted Control Engineering," *Proceedings CACSD'92*, IEEE Computer–Aided Control System Design Conference, Napa, Calif.
4. Berkan, R.C., B.R. Upadhyaya, L.H. Tsoukalas, R.A. Kisner, and R.L. Bywater, (1991). "Advanced Automation Concepts for Large–Scale Systems," *IEEE Control Systems Magazine*, October, pp. 4–12.
5. Booch, G., (1991). *Object–Oriented Design with Applications*, Benjamin/Cummings, Redwood City, Calif.
6. Cellier, F.E., (1991). *Continuous System Modeling*, Springer–Verlag, New York.
7. Cellier, F.E., (1992). "Integrated Continuous–System Modeling and Simulation Environments," in: *CAD for Control Systems* (D. Linkens, Ed.), Marcel Dekker, New York, in press.
8. Cellier, F.E., P.O. Grepper, D.F. Rufer, and J. Tödtli, (1977). "AUTLIB, Automatic Control Library, Educational Aspects of Development and Application of a Subprogram Package for Control," *Proceedings IFAC Symposium on Trends in Automatic Control Education*, Barcelona, Spain, March 30 – April 1, Pergamon Press, Oxford, U.K., pp. 151–159.
9. Cellier, F.E., and C.M. Rimvall, (1987). "Computer–Aided Control System Design: Techniques and Tools," in: *Systems Modeling and Computer Simulation* (N. Kheir, Ed.), Marcel Dekker, New York, pp. 631–679.
10. Cellier, F.E., L.C. Schooley, B.P. Zeigler, A. Doser, G. Farrenkopf, J. Kim, Y. Pan, and B. Williams, (1992). "Watchdog Monitor Prevents Martian Oxygen Production Plant from Shutting Itself Down during Storm," *Proceedings ISRAM'92, International Symposium on Robotics and Manufacturing*, Santa Fe, N.M., November 8–11, 1992.
11. Chi, S.D., (1991). *Modelling and Simulation for High Autonomy Systems*, Ph.D. Dissertation, Department of Electrical and Computer Engineering, University of Arizona, Tucson, Ariz.
12. Chi, S.D., B.P. Zeigler, and F.E. Cellier, (1990). "Model–Based Task Planning System for a Space Laboratory Environment," *Proceedings SPIE Conference on Cooperative Intelligent Robotics in Space*, Boston, Mass.

13. CSTOL, (1991). *OASIS CSTOL Reference Guide*, University of Colorado at Boulder, Operations and Information Systems Group, Laboratory for Atmospheric and Space Physics, Boulder Colo.
14. Dongarra, J.J., (1979). *Linpack Users' Guide*, SIAM, Philadelphia.
15. Doyle, J.C., B.A. Francis, and A.R. Tannenbaum, (1992). *Feedback Control Theory*, Macmillan Publishing, New York.
16. Earl, A., (1990). *A Reference Model for Computer Assisted Software Engineering Environment Frameworks*, Technical Report HPL–SEG–TN–90–11, Software Environments Group, Hewlett–Packard Laboratories, Bristol, U.K.
17. Gavel, D.T., and C.J. Herget, (1984). *The M Language — An Interactive Tool for Manipulating Matrices and Matrix Ordinary Differential Equations*, Internal Report, Signal and Image Processing Research Group, Lawrence Livermore National Laboratory, University of California, Livermore, Calif.
18. Garbow, B.S., J.M. Boyle, J.J. Dongarra, and C.B. Moler, (1977). *Matrix Eigensystem Routines — Eispack Guide Extension*, Springer–Verlag, New York, Lecture Notes in Computer Science, **51**.
19. Hack, B.W.J., (1988). *Man to Machine, Machine to Machine and Machine to Instrument Interfaces for Teleoperation of a Fluid Handling Laboratory*, MS Thesis, Technical Report TSL–014/88, Electrical and Computer Engineering Department, University of Arizona, Tucson Ariz.
20. Helgert, H.J., (1991). "Services, Architectures, and Protocols for Space Data Systems," *Proceedings of the IEEE*, **79**(9), pp. 1213–1231.
21. Heller, D., (1991). *Motif Programming Manual*, O'Reilly & Associates, Sebastopol, Calif.
22. Integrated Systems, Inc., (1984). *MATRIX$_X$ User's Guide, MATRIX$_X$ Reference Guide, MATRIX$_X$ Training Guide, Command Summary and On–Line Help*, Santa Clara, Calif.
23. ISO, (1983). *Basic Reference Model for Open Systems Interconnection*, Technical Report DIS 7498, International Organization for Standardization.
24. Kury, P.M., (1990). *An Intelligent Fault Diagnoser for Distributed Processing in Telescience Applications*, MS Thesis, Department of Electrical and Computer Engineering, University of Arizona, Tucson, Ariz.
25. Lew, A.K., (1988). *Astrometric Telescope Simulator for the Design and Development of Telescope Teleoperation*, MS Thesis, Technical Report TSL–016/88, Electrical and Computer Engineering Department, University of Arizona, Tucson Ariz.
26. Mathworks, Inc., (1992). *The Student Edition of MATLAB for MS-DOS or Macintosh Computers*, Prentice–Hall, Englewood Cliffs, N.J.
27. Moler, C.B., (1980). *Matlab User's Guide*, Dept. of Comp. Sc., University of New Mexico, Albuquerque.
28. Motaabbed, A., (1992). *A Knowledge Acquisition Scheme for Fault Diagnosis in Complex Manufacturing Processes*, MS Thesis, Dept. of Electr. & Comp. Engr., University of Arizona, Tucson, Ariz.
29. NASA, (1985). *The Space Station Project.*

30. OASIS, (1991). *OASIS System Manager's Guide*, University of Colorado at Boulder, Operations and Information Systems Group, Laboratory for Atmospheric and Space Physics, Boulder Colo.

31. Rasmussen, J., (1990). "Supervisory Control and Risk Management," in: *Robotics, Control and Society*, (N. Moray, W.R. Ferrell, and W.B. Rouse, Eds.), Taylor & Francis, New York, pp. 160–171.

32. Rimvall, C.M., (1986). *Man-Machine Interfaces and Implementational Issues in Computer-Aided Control System Design*, Ph.D. Dissertation, Diss. ETH No 8200, Swiss Federal Institute of Technology, ETH Zürich, Switzerland.

33. Sarjoughian, H.S., F.E. Cellier, and B.P. Zeigler, (1990). "Hierarchical Controllers and Diagnostic Units for Semi-Autonomous Teleoperation of a Fluid Handling Laboratory," *Proceedings IEEE Phoenix Conference on Computers and Communication*, Scottsdale, Ariz., pp. 795–802.

34. Sheridan, T.B., (1988). "Supervisory Control of Telerobots in Space," in: *Machine Intelligence and Autonomy for Aerospace Systems* (E. Heer and H. Lum, Eds.), Progress in Astronautics and Aeronautics, **115**, American Institute of Aeronautics and Astronautics, pp. 31–50.

35. Strickland, T.J., (1989). *Dynamic Management of Multi-Channel Interfaces for Human Interaction with Computer-Based Intelligent Assistants*, Ph.D. Dissertation, Dept. of Systems & Industrial Engr., University of Arizona, Tucson, Ariz.

36. Systems Control Technology, Inc., (1985). *CTRL-C, A Language for the Computer-Aided Design of Multivariable Control Systems, User's Guide*, Palo Alto, Calif.

37. TAE+, (1991). *TAE+ User Interface Developer's Guide*, Version 5.1, National Aeronautics and Space Administration (NASA), Goddard Space Flight Center, distributed by: Cosmos, University of Georgia, Athens, Ga.

38. Vanbegin, M., and P. Van Dooren, (1985). *MATLAB-SC*, Technical Note N168, Philips Research Laboratories, Bosvoorde, Belgium.

39. Wang, Q., and F.E. Cellier, (1991). "Time Windows: An Approach to Automated Abstraction of Continuous-Time Models into Discrete-Event Models," *International Journal of General Systems*, **19**(3), pp. 241–262.

40. Young, D.A., (1989). *X Window Systems Programming and Applications With Xt*, Prentice-Hall, Englewood Cliffs, N.J.

41. Zeigler, B.P., S.D. Chi, and F.E. Cellier, (1991). "Model-Based Architecture for High Autonomy Systems," in: *Engineering Systems with Intelligence — Concepts, Tools and Applications* (S.G. Tzafestas, Ed.), Kluwer Academic Publisher, Dordrecht, The Netherlands, pp. 3–22.