WATCHDOG MONITOR PREVENTS MARTIAN OXYGEN PRODUCTION PLANT FROM SHUTTING ITSELF DOWN DURING STORM

F.E. CELLIER, L.C. SCHOOLEY, B.P. ZEIGLER A. DOSER, G. FARRENKOPF, J. KIM, Y. PAN and B. WILLIAMS

Dept. of Elect. & Comp. Engr., University of Arizona
Tucson, Arizona 85721, U.S.A.
E_mail: Cellier@ECE.Arizona.Edu

ABSTRACT

A distributed intelligent controller was built for high-autonomy operation of a prototype of a Martian Oxygen Production Plant. During a 100 hour test run conducted to verify the reliability of the high-autonomy control architecture, the controller broke down in the consequence of a lightning stroke. The intelligent controller was able to recognize that a fault had occurred, isolate the fault by recognizing that the fault was not to be found in the plant but had to be in the controller itself, and was able to take corrective action. The controller was able to correct this fault, although the type of fault that had occurred was not previously foreseen. This paper outlines the intelligent control architecture, describes the incident, and summarizes the lessons learned.

INTRODUCTION

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 1 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.1).

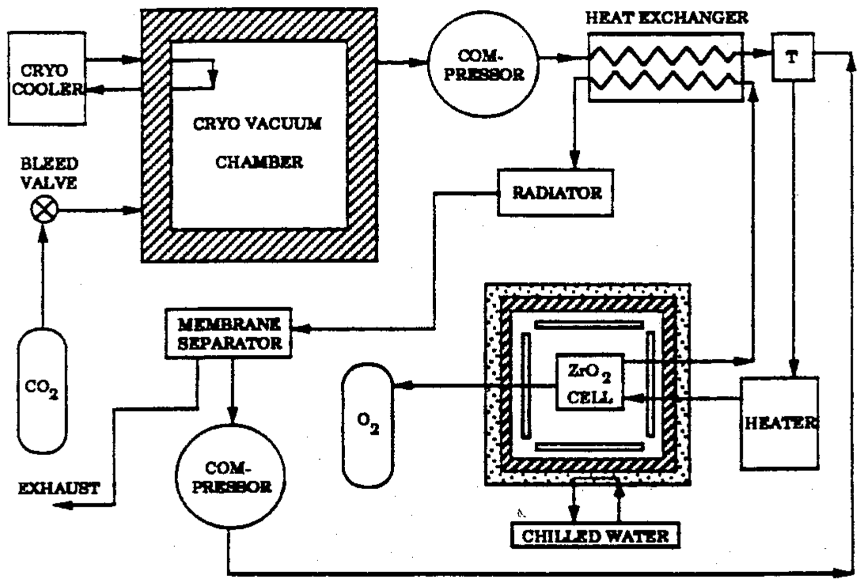

Figure 1: Martian Oxygen Production Plant

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.

CONTROL ARCHITECTURE FOR HIGH–AUTONOMY OPERATION

According to NASA [4], *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.

Traditional control engineers hardly ever concern 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 a control engineer. He (or she) is 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 an additional hierarchy layer about the control architecture. It usually contains several alternative control architectures. 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.

An architecture for high–autonomy systems was recently proposed by Zeigler *et al.* [7] and Chi [1]. The oxygen production plant prototype, while still fairly primitive, contains all elements of a high–autonomy system. In the sequel, these elements will be described, and it will be shown how they cooperated during the incident.

Task Planning

Task planning refers to the activity of sequencing commands issued at task level to the control architecture. The task planner does not concern itself with the details of task execution. All it does is to determine which tasks should be executed, when, and in what sequence; it concerns itself with *resource management*, i.e., it knows which resources are required by each task, and makes sure that those resources are available during task execution; and finally, it concerns itself with *time management*, i.e., it knows how much time each task is supposed to consume and makes sure that real–time constraints are satisfied.

In this testbed, the task planner was fairly rudimentary. Since only one type of experiment was planned (the 100 hour test of continuous oxygen production without human intervention), an elaborate task planner was not required. The currently used task planner is 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.

Chi *et al.* [2] and Chi [1] have proposed an intricate hierarchical task planning architecture that should suffice for controlling such an expedition.

Command Execution

The command executor accepts the next command to be executed from the task planner and prepares the control architecture for its execution. The command executor selects the appropriate low–level control algorithm from a set of precoded algorithms, downloads the selected controller into the smart sensor that physically houses the low–level control loop, and initiates the control activity.

The command executor 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 executor is responsible for fault detection during transient operational phases, such as the start–up and shut–down phases. 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. The time–window mechanism has been described in detail by Wang and Cellier [6].

Plant Operation

The actual low–level control is implemented in a microcontroller called a smart sensor. Once the command executor has downloaded a control program to the smart sensor, the low–level controller is activated. This can be a classical controller of any vintage. In this testbed, all control activity was strictly 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.

Watchdog Monitors

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 by Kury [3]. In the described incident, 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.

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 this 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 asserted 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 is needed. Such an architecture was first proposed by Sarjoughian *et al.* [5]. It has meanwhile been elaborated upon by Chi [1].

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.

THE INCIDENT

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

Figure 2 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.

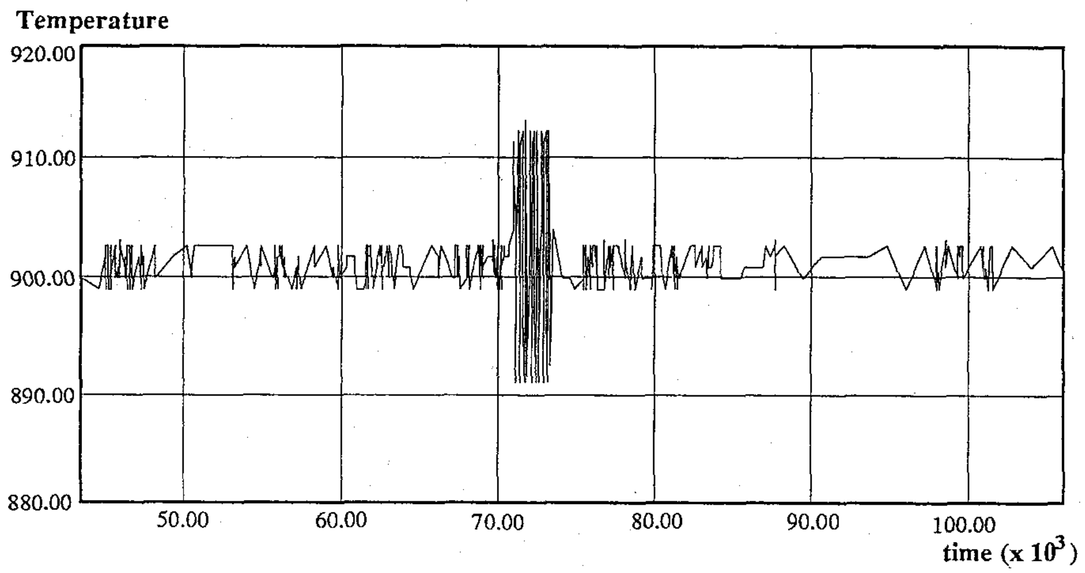

Figure 2: ZrO_2 Temperature Plotted over Time

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

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, a short power failure (less than 2 seconds) occurred. The backup power supply took over, but there are signs of a temporary power surge. This transient upset the smart–sensor–based temperature controller; more precisely, the real–time clock of the smart sensor stopped operating.

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, no watchdog monitor suspected any trouble. After a long time, a watchdog responsible for checking temperature values got suspicious, but this happened 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 implementation.

Fault diagnosers are harmless. They are sequential routines and don't consume any computing cycles unless an anomaly has been observed. In the reported 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 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.

LESSONS LEARNED

The lessons learned from this incident are enumerated in the sequel:

1. Proper functioning of the real–time clock is crucial to any event–based controller. Consequently, the smart sensor should report the value of its real–time clock back to the PC, and an additional watchdog should be installed that compares the two real–time clocks with each other and gets aroused when they start to diverge.
2. Some of the events that do not normally occur during steady–state operation can be declared as anomalous events. While no separate immediate recovery action may be needed, all anomalous events should nevertheless trigger a fault diagnoser.
3. While either of the two previous lessons could have improved results in the given situation, the watchdog that finally caught on should have been executed a little more frequently.
4. Event–based control is well suited for post–fault analysis. However, to this end it is necessary to remember which rules were fired when, why, and by whom. This was omitted in order to save computing cycles. Due to this oversight, it wasn't possible to conclude without a grain of a doubt what really had happened during the incident.

5. Similarly, while the smart sensor is equipped with means to download into it programs on the fly, nobody ever thought of the need to upload a program from the smart sensor back into the PC to save it for post–fault analysis.
6. The lessons learned from this incident should ensure that the same incident will never happen again. These sorts of experiments will help increase the reliability of the high–autonomy operation until a system results that is reliable enough for long–term operation on planet Mars. However, a good amount of research is still needed before this ultimate goal can be reached.

SUMMARY AND CONCLUSIONS

This paper outlined an intelligent high–autonomy control architecture useful for robots and manufacturing plants. The advocated methodology, once consolidated, will prove particularly valuable for space exploration in the early days of the next century, but it will also be useful for other applications where human presence at the plant location may be either undesirable, or hazardous, or impossible. Of particular interest was the fact that the architecture was able to detect and counter an anomaly in the controller itself, the potential occurrence of which had not been anticipated.

REFERENCES

1. Chi, S., (1991). *Modelling and Simulation for High Autonomy Systems*, Ph.D. Dissertation, Department of Electrical and Computer Engineering, University of Arizona, Tucson, Ariz.
2. Chi, S., 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.
3. 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.
4. NASA, (1985). *The Space Station Project.*
5. 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.
6. 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.
7. Zeigler, B.P., S. Chi, and F.E. Cellier, (1991). "Model–Based Architecture for High Autonomy Systems," *Proceedings EURISCON'91 — European Robotics and Intelligent Systems Conference*, Corfu, Greece, June 23–28.