NASA Technical Memorandum 102412
AIAA-90-0744

Flight Software Development for the Isothermal Dendritic Growth Experiment

Laurie H. Levinson and Edward A. Winsa
Lewis Research Center
Cleveland, Ohio

and

Martin E. Glicksman
Rensselaer Polytechnic Institute
Troy, New York

Prepared for the
28th Aerospace Sciences Meeting
sponsored by the American Institute of Aeronautics and Astronautics
Reno, Nevada, January 8-11, 1990

NASA

FLIGHT SOFTWARE DEVELOPMENT

FOR THE

ISOTHERMAL DENDRITIC GROWTH EXPERIMENT

Laurie H. Levinson, Edward A. Winsa, and Martin E. Glicksman*
NASA Lewis Research Center, Cleveland, Ohio
*Rensselaer Polytechnic Institute, Troy, New York

ABSTRACT

The Isothermal Dendritic Growth Experiment (IDGE) is a microgravity materials science experiment scheduled to fly in the cargo bay of the shuttle on the United States Microgravity Payload (USMP) carrier. The experiment will be operated by real-time control software which will not only monitor and control onboard experiment hardware, but will also communicate, via downlink data and uplink commands, with the Payload Operations Control Center (POCC) at NASA George C. Marshall Space Flight Center (MSFC). The software development approach being used to implement this system, which will be the focus of this paper, began with software functional requirements specification. This was accomplished using the Yourdon/DeMarco methodology as supplemented by the Ward/Mellor real-time extensions. The requirements specification in combination with software prototyping was then used to generate a detailed design consisting of structure charts, module prologues, and Program Design Language (PDL) specifications. This detailed design will next be used to code the software, followed finally by testing against the functional requirements. The result will be a modular real-time control software system with traceability through every phase of the development process.

INTRODUCTION

The Isothermal Dendritic Growth Experiment (IDGE) is a microgravity materials science experiment currently planned for three flights beginning in 1993. It is scheduled to fly on the United States Microgravity Payload (USMP) carrier located in the cargo bay of the Space Shuttle (Figure 1). The experiment, originally proposed by Professor M. E. Glicksman - now the Principal Investigator - of Rensselaer Polytechnic Institute, is being designed and built at the NASA Lewis Research Center (LeRC) in Cleveland, Ohio.[1,2]

The scientific objective of the IDGE is to test current mathematical models which predict dendrite growth velocity and tip radius in a solidifying metal melt as functions of dendrite physical properties and metal melt properties. The data being gathered will provide the means to verify the existing models or, in the event these models should prove flawed, to correct them. Correct models could lead ultimately to improved techniques for commercial metal production, since virtually all metals and alloys solidify from the molten state by a dendritic process.

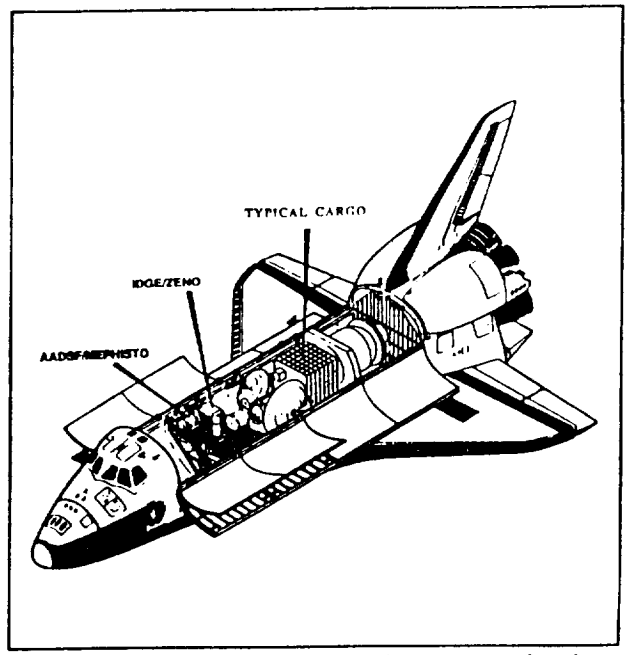

Figure 1. The IDGE, mounted on the USMP carrier in the Space Shuttle cargo bay and shown along with its mission payload complement.

The IDGE technical objective is to build and fly an apparatus that meets all IDGE scientific requirements, as well as all Space Shuttle safety and interface requirements. In terms of the software development effort, this implies the development of a system which will operate the experiment autonomously, will recover automatically from most system faults, will accept input data from the Space Acceleration Measurement System (SAMS), will communicate with the Payload Operations Control Center (POCC) both via downlink

data and uplink commands, and will be easily maintainable over the lifetime of the experiment (nominally, three flights). Autonomous operation of the experiment will, in this case, include such features as precise temperature control, onboard image analysis, 35 mm camera control, inter-processor communication among three onboard processors, and onboard storage of all critical science data.

OVERVIEW

The Software Development Life Cycle

When developing a complex software system such as that described above for the IDGE, the development process is commonly broken down into five basic stages: analysis, design, coding, testing, and maintenance. Because these five stages cover the entire range of activities that may take place over the lifetime of a software system, they are, together, referred to as the software development life cycle. Of the five stages which make up this life cycle, the first four relate to initial system development, while the final stage - maintenance - refers to any changes which must be made to the system following its initial completion. Maintenance aside, it is generally recommended that approximately 50% of the available resources (i.e., time and money) be allocated to analysis and design, approximately 15% to coding, and approximately 35% to testing.

Analysis

The goal of the analysis phase is to obtain agreement among all relevant parties as to the functional requirements of the software system. This can, alternatively, be thought of as the time during which the "what" of the system is specified (in other words, what the system is to do).

In the case of a space experiment such as the IDGE, functional requirements specification is accomplished using input both from previously generated documents (e.g., the Science Requirements Document and the Engineering Requirements Document) and from discussions with science and engineering team members. These inputs are then used to generate a Software Functional Requirements Document, which is the primary output of the analysis phase.

On the IDGE project, software functional requirements were specified using the Yourdon/DeMarco methodology, as supplemented by the Ward/Mellor real-time extensions. This methodology will be described in a subsequent section of this paper.

Design

The design phase is the time during which the "how" of the software system is specified. Using the output of the analysis phase as input, the system designer determines precisely how the required software functions will be implemented. The resulting detailed design is specified in a Software Design Document, which is the primary output of the design phase.

On the IDGE project, in addition to the input provided by the Software Functional Requirements Document, information gained from software prototyping is being used as input to the design process. The reason for this, as well as a description of the specific design approach used on the project, will be provided below.

Coding

During the coding stage, the previously generated detailed design is used to produce the actual system code which the computer hardware will execute. While this task may be accomplished by the system designer him/herself, it can also be done by someone else entirely - a coder - provided the system has been developed so as to permit such an approach.

On the IDGE project, engineering code which is based on the flight system design but written by someone other than the system designer is being used to test the engineering hardware. While the actual flight code is to be written by the system designer, for purposes of testing engineering hardware, this alternative approach has been quite successful. This experience, as well as the circumstances that are conducive to using such an approach, will be discussed below.

Testing

The testing phase always consists of two distinct types of testing: unit testing and integration testing. In addition, if software is being delivered to a customer, a third type of testing - acceptance testing - is performed by the customer. These three types of testing are performed in the order mentioned, and an error found at any point in the process necessitates a return to unit testing once the error has been corrected.

Unit testing refers to the testing performed individually on each unit of the software system. During this testing, the goal is to ensure that every path through a unit's code has been executed and been shown to provide the desired result.

Integration testing refers to the testing performed on the entire system, or on some interconnected set of units in the system, once the individual units have been tested. The purpose of this type of testing is to ensure that the system as a whole functions as expected. Integration testing is thus functional testing designed to demonstrate that required system capabilities operate as desired.

The methods used to do unit testing and integration testing on the IDGE project will be discussed below.

Acceptance testing, as mentioned above, refers to the testing done once the software system has been delivered to the customer. On the IDGE project, however, software is not actually being delivered to a customer, so all testing activities will be carried out during unit and integration testing. Acceptance testing, therefore, will not be discussed further.

Maintenance

The maintenance phase of the software development process covers the entire period of time following initial system completion. Any change made to the software during this time, regardless of its nature, is considered part of this phase of development. Maintenance, therefore, includes not only changes made to fix a previously undetected bug, but also application-specific changes made to customize a system (e.g., in the case of a space experiment, mission-specific changes) or upgrades made due to changes in technology.

While the impact that the IDGE software development approach has on the maintenance process will be discussed, a discussion of the maintenance process itself is beyond the scope of this paper.

FUNCTIONAL REQUIREMENTS SPECIFICATION

System functional requirements specification is the first, and perhaps the most critical, step in the software development process. In generating the specification, the system developer must communicate with all those involved in the project and establish the goals for the remainder of the development effort. In order to accomplish this task in an effective manner, it is important that the method used to generate the specification possess certain characteristics.

Since user-developer communication is a major concern at this stage of the development process, the method chosen should facilitate this communication and maximize the potential for users to provide input to the developer. In order to do so, the method selected should (1) use a vocabulary with which the users are familiar, and (2) present a "condensed" version of the system, where certain details are suppressed in favor of presenting "the big picture." This latter quality will not only maximize the probability of users finding requirements errors during the review process, but will also allow the developer to take an incremental approach to development. Such an approach simplifies the system developer's task by allowing him/her to take advantage of varying degrees of abstraction. At the same time, it provides the ability to compare the system description at various stages and check for consistency, thereby helping to ensure proper development.

In addition to the above features, a system specification approach should also provide ease of maintenance, so that any necessary changes can be made without difficulty. This is an important practical consideration, since change is an inherent part of the specification process.

Furthermore, while the approach used should present "the big picture," as mentioned above, it should also permit the system to be described in sufficient depth. This can be accomplished by using an approach which takes advantage of partitioning and leveling of detail.

Finally, one additional desirable feature - which might be considered a consequence of those mentioned thus far - is that the approach used should be primarily graphical as opposed to verbal. A graphical approach is not only consistent with the previously mentioned features but, in fact, provides an excellent means of implementing those features.

The Specification Methodology

The type of approach used on the IDGE project to accomplish system functional requirements specification is described in detail in References 3 and 4, and possesses all of the above-mentioned desirable characteristics. This approach is referred to as the Yourdon/DeMarco methodology with Ward/Mellor real-time extensions.

This method of system specification involves the development of an essential model, so called because it separates the essence of a system from its implementation. The essential model describes what a system does and what data it stores irrespective of the technology used.

As with any model, certain assumptions are made as part of the modeling process. In this case, three basic assumptions apply: (1) The technology is assumed to be perfect, which implies no internal system errors are generated; (2) the processor is assumed to have infinite memory, which implies there are no storage concerns; and (3) the processor is assumed to have infinite processing capacity, which implies that processes run in zero time. By making these three assumptions, all implementation aspects of the problem are effectively removed from consideration.

In generating the essential model, three types of diagrams may be used: (1) data flow diagrams, (2) state transition diagrams, and (3) entity relationship diagrams.

Data flow diagrams (DFDs), as the name indicates, show the flow of data in the system. The IDGE essential model consists of twenty-one DFDs.

State transition diagrams (STDs) show the different states in which the system may be found and describe the transition from one state to another. Twenty STDs were required to define the IDGE essential model.

Entity relationship diagrams are used to show the organization of data in the system, and are most useful in a system which is very data-intensive. The IDGE software system is control-oriented rather than data-oriented, however, and is thus not a data-intensive system. Consequently, no entity relationship diagrams were produced. This aspect of the essential modeling process will, therefore, not be discussed any further. The reader is referred to Reference 4 should additional information on the topic be desired.

The IDGE Essential Model

The essential model used to define the system functional requirements is actually composed of two distinct models: the environmental model, which describes the environment in which the system operates, and the behavioral model, which describes the behavior of the system.

The Environmental Model. The IDGE environmental model consists of two items: (1) the context diagram, a special type of data flow diagram which, as the name implies, represents the context in which the software system operates; and (2) the event list, which is a list of all the events that occur in the environment to which the system will have a pre-planned response.

The IDGE context diagram is shown in Figure 2. The large circle in the center of the diagram represents the software system, while the boxes surrounding it represent all the external subsystems with which the software communicates. The labeled arrows between the two indicate the net data flows into or out of the software system.

A portion of the IDGE event list is shown in Figure 3. The external event is listed on the left, and the pre-planned system response is listed on the right.

The Behavioral Model. The behavioral model for the IDGE software system consists of three components: (1) leveled data flow diagrams, (2) state transition diagrams, and (3) a data dictionary.

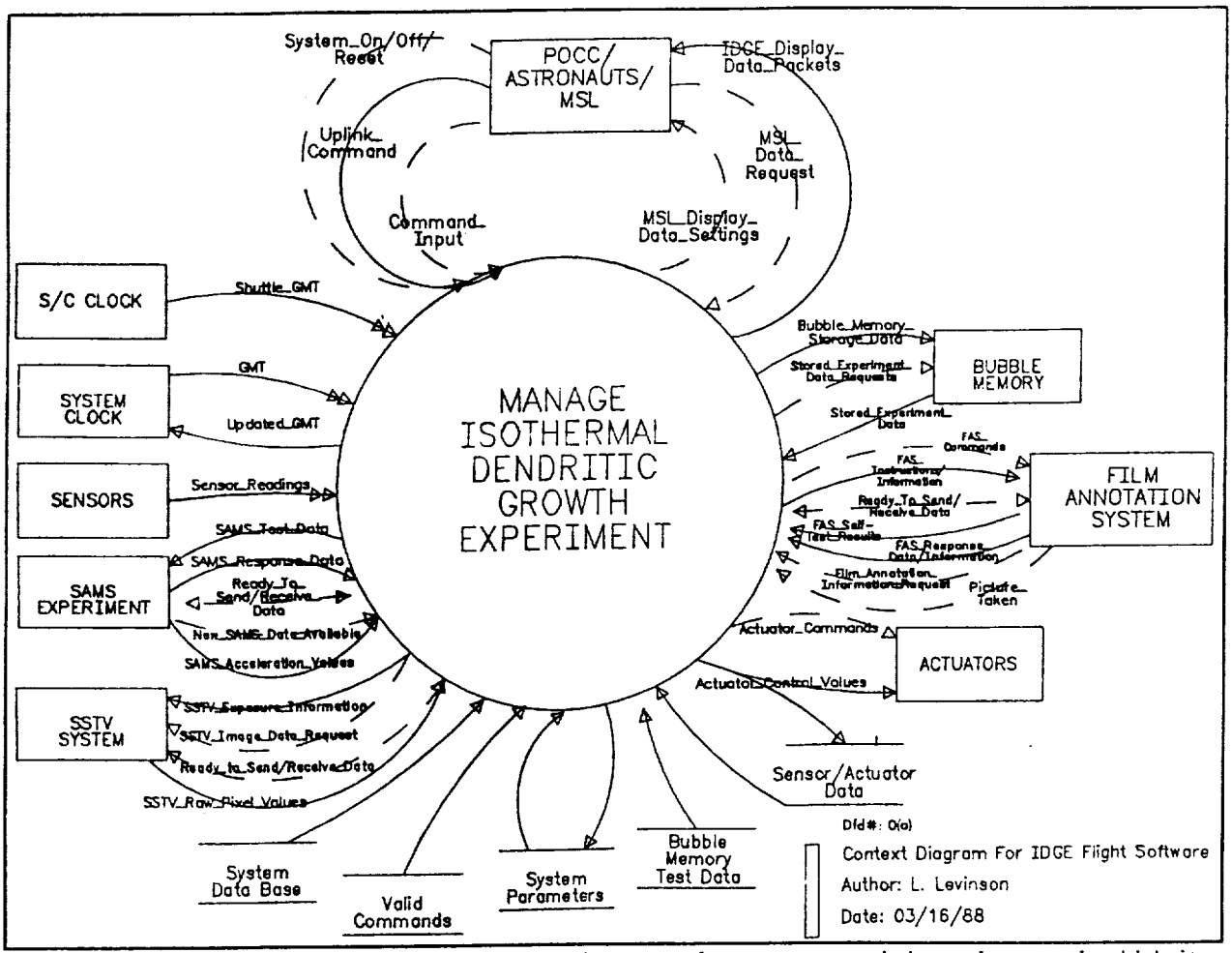

Figure 2. The IDGE context diagram, which depicts the IDGE software system and the environment in which it operates.

EVENT LIST

E: — Enable
D: — Disable
T: — Trigger

	Event	System Response
1.	Astronaut turns on/ resets IDGE	Set System_Status_Flag to Initializing_System E: Send Data to POCC E: Compare and Update System Clocks
2.	New SAMS data available	Reset Max_Time_to_Wait_for_SAMS_Data Timer T: Collect and Update SAMS Data
3.	MSL requests IDGE data	T: Output IDGE Data Packet
4.	Astronaut/POCC sends command	T: Process Command Input
5.	Sensor readings indicate SCN stable at calibration temperature	T: Calibrate Thermistors E: Perform Pause Command Processing E: Initiate New Cycle
6.	Sensor readings indicate bath stable at melt temperature	Set Time_to_Stay_at_Melt_Temperature Timer
7.	Time to stay at melt temperature has elapsed	Set Target_Temperature to Cycle_Supercool_ Temperature Lower: Stable_Bath_Temperature_Flag T: Update Cycle Parameters E: Operate Blowers E: Determine Bath Temperature Stabilization
8.	Sensor readings indicate SCN stable at supercool temperature	T: Take Picture of Stinger E: Initiate Dendritic Growth Set Time_to_Begin_Image_Analysis Timer
9.	Time to begin image analysis occurs	E: Analyze SSTV Image Data Set Max_Time_to_Do_Image_Analysis Timer
10.	SSTV images indicate dendrite growth has occurred or Max_Time_ to_Do_Image_Analysis timer has expired	D: Initiate Dendritic Growth D: Analyze SSTV Image Data Set Time_to_Initiate_Photo_Sequence Timer
11.	Time to take SSTV image to send to POCC	T: Take Image

Figure 3. A portion of the IDGE event list.

There are twenty leveled DFDs which describe the IDGE software system. One of these is shown in Figure 4. The solid circles on the diagram are referred to as data transformations, because they show the manner in which data is transformed by the system. The dashed circle in the center of the diagram is referred to as a control transformation because it contains the logic that controls the processing. That logic is specified in detail using a state transition diagram.

On the state transition diagram, Figure 5, the boxes represent the different states in which the system can be found. The items listed between each pair of states give the conditions which will cause the system to change states - shown above the line - and the actions that will occur during the transition - shown below the line.

The final component of the behavioral model, the data dictionary, lists and defines all data flows shown on the DFDs. The IDGE data dictionary contains approximately 275 entries and uses the notational conventions described in Reference 1. A sample page from this dictionary is shown in Figure 6.

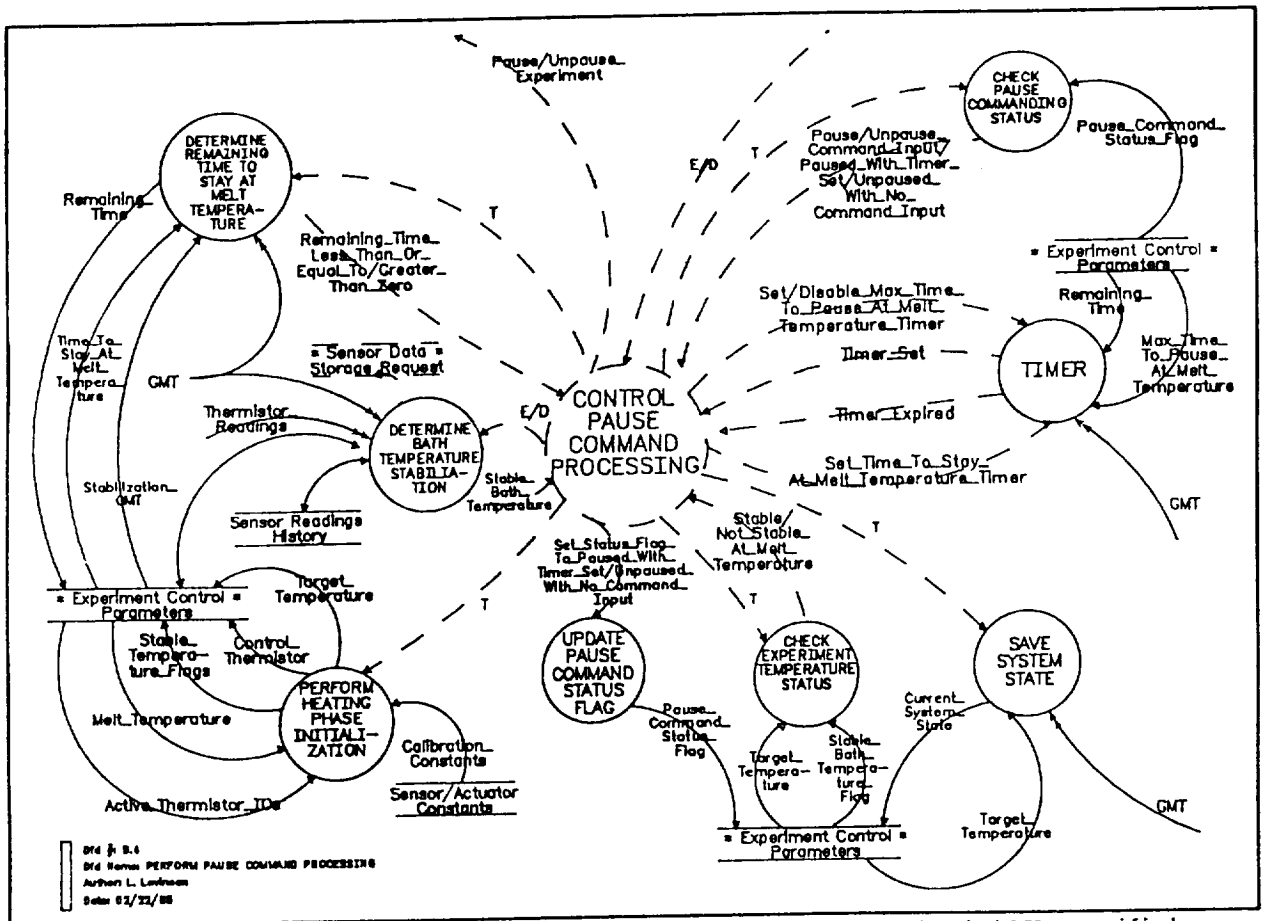

Figure 4. The IDGE DFD describing the pause command processing, one of twenty leveled DFDs specified.

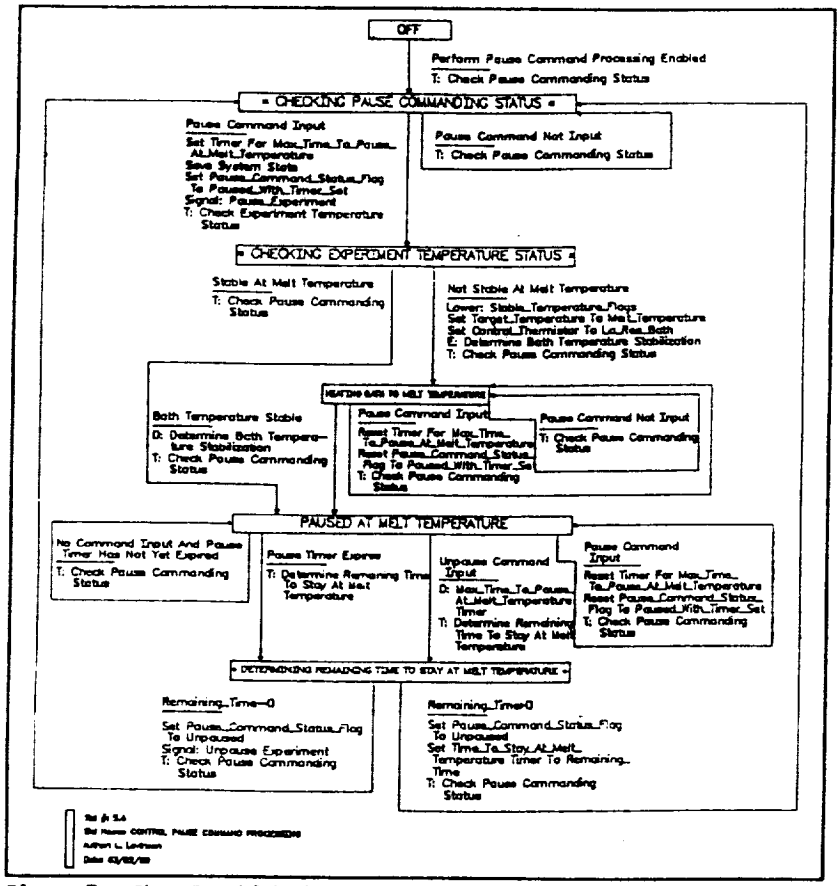

Figure 5. The STD which describes the pause command processing control logic.

Development of the IDGE Essential Model

In developing the IDGE essential model, the first step in the process was to generate a context diagram which showed the IDGE system in relationship to the various subsystems in the environment with which it would communicate.

The next step in the process was then determined by consideration of the type of system being developed. As the primary purpose of the IDGE software system is to control the responses to various external events, this system is considered a control-dominated system. When developing the essential model for such a system, the next step following context diagram preparation is generation of an event list.

Once the event list has been generated, this list is used to guide the preparation of the state transition diagrams. The event list not only helps to determine what STDs are needed, but also helps to illuminate the interconnections between STDs.

Once the STDs have been generated, they are then used in conjunction with the context diagram to prepare a set of leveled DFDs. As the DFDs are prepared, all data flows on the diagrams are added to the data dictionary and defined.

To complete the IDGE essential model in the above-described manner required the full-time effort of one software engineer for approximately nine months. At the conclusion of this nine month period, a final draft of the Software Functional Requirements Document was issued. This was subsequently signed by the primary users, indicating satisfaction with its content.

DESIGN

The purpose of the design phase, as previously indicated, is to elaborate on the functional requirements specification in such a manner as to describe precisely how the system requirements will be implemented. As with the requirements specification approach, it is highly desirable that the design methodology chosen permit comprehension of the system as a whole and, at the same time, provide the ability to describe the system in a sufficiently detailed and in-depth manner. As mentioned above, a methodology which takes advantage of partitioning and leveling of detail has the ability to do just that.

One such methodology is that recommended by Ward and Mellor in Reference 4. This approach consists of using the essential model generated during the analysis phase to develop what is referred to as an implementation model. While the essential model is technology-independent, the implementation model describes the system as it is actually realized by a specific technology.

Development of this implementation model involves the generation of three distinct models: the processor model, the task model, and the module model.

The processor model describes the allocation of the previously defined processes to individual processors, along with any inter-processor interfaces which result from such an allocation. This model is derived directly from the essential model, and thus results in a portion of the essential model being contained in each processor model.

The task model describes the allocation of each process shown on the processor model to individual tasks, as well as any resulting inter-task interfaces. This model is therefore a further refinement of the processor model.

Both the processor model and the task model are implemented using data flow diagrams.

"Send_Test_Command"_Message =	Message sent to the POCC and displayed on the IDGE Display which indicates that the IDGE software is ready to accept the Test_Command
Sensor/Actuator_Constant =	[Power_Equation_Constants | Bath_Heater_Resistance | Calibration_Constants | Thermistor_Amplifier_Offset_Voltage | Expected_Minimum_Rate_of_Heating]
Sensor/Actuator_Constants =	Storage area containing the value of each Sensor/Actuator_Constant
Sensor/Actuator_Data =	Actuator_Control_Values + Sensor_Readings_History + Sensor/Actuator_Constants
Sensor_Data_Storage_Request =	Intermediate storage for the Time_to_Store_Sensor_Data flag
Sensor_Reading =	[Bath_Cooling_Duct_Blower_Status_Flag | Bath_Heater_Current_Reading | Thermistor_Reading | Spot_Cooler_Status_Flag | Thermostat_Fluid_Pressure]
Sensor_Readings_History =	Storage area containing the most recent Bath_Cooling_Duct_Blower_Status_Flag value, Bath_Heater_Current_Reading, Spot_Cooler_Status_Flag value, Thermistor_Readings, Thermistor_Reading_GMTs, Thermostat_Fluid_Pressure, Averaged_Thermistor_Readings, and Averaged_Reading_GMTs.
Shell_Thermistor_Reading =	Sensor_Reading provided by a low resolution thermistor and used to determine the shell temperature
Shuttle_GMT =	GMT indicated by the Shuttle clock
Spot_Cool_Time =	Maximum amount of time that the spot cooler should remain on
Spot_Cooler_On/Off =	Actuator_Command which causes the spot cooler to be turned on or off
Spot_Cooler_Status_Flag =	Sensor_Reading which indicates the on/off status of the spot cooler
Spot_Cooler_Switch_Setting =	Actuator_Control_Value which indicates the spot cooler switch setting

Figure 6. Sample page from the IDGE data dictionary.

The module model describes the allocation of system activities to modules and shows the hierarchical organization of these modules. The graphical portion of this model is implemented using a type of diagram referred to as a structure chart. Associated with each structure chart is a set of module specifications, which provide a more detailed description of each module on the structure chart. A module specification can take a variety of forms, but is usually verbal, and thus constitutes a non-graphical portion of the model.

IDGE Design Approach

On the IDGE project, the software design is being implemented using only the module model portion of the Ward/Mellor design methodology. There are basically two reasons for taking this approach.

First, the allocation of system activities to the different processors was already decided at the time the essential model was constructed, and this information could therefore be taken into account as the essential model was developed. As a result, the completed essential model provided an accurate description which, for a significant portion of the system, would require few changes in order to produce the implementation model. Those portions not included in or adequately represented by the essential model seemed to fall into one of two categories: Either they were considered to be at a low enough level to be handled exclusively in the context of the module model; or they were felt to be so complex as to require a more "applied" approach, i.e., software prototyping. As a result, it was decided that relatively little benefit would be obtained by taking the time to generate the processor and task models.

The second reason for using this approach is purely practical in nature: The IDGE flight software is being implemented on a three-processor system by one software engineer. In order to generate the processor and task models, three complete sets of data flow diagrams would have to be generated for each model: one set for each processor. This would be far too time-consuming a task for one software engineer, particularly given the tight flight schedule and the relative lack of benefit anticipated, as mentioned above. In lieu of this, any complex subsystems not modeled by the essential model, such as inter-processor communication among the three onboard processors, are being handled by developing prototype software. Given the particular situation, this is felt to be a much more efficient and beneficial use of the available time.

By the same token, however, it should be mentioned that use of all three design models suggested by Ward and Mellor might be much more critical and/or far more beneficial on a different project, particularly a larger one. On such a project, multiple programmers/software engineers would, for example, not only make it more feasible to generate the increased documentation, but could also create a much greater need for interfaces to be explicitly specified.

Development of the IDGE Implementation Model

The IDGE implementation model consists of three elements: structure charts (SCs), module prologues, and Program Design Language specifications (PDLs).

The structure chart, Figure 7, provides a description of the hierarchical structure of the software system. It not only shows what module invokes which lower level modules (i.e., "who calls who"), but also the input and output data passed between each pair of modules.

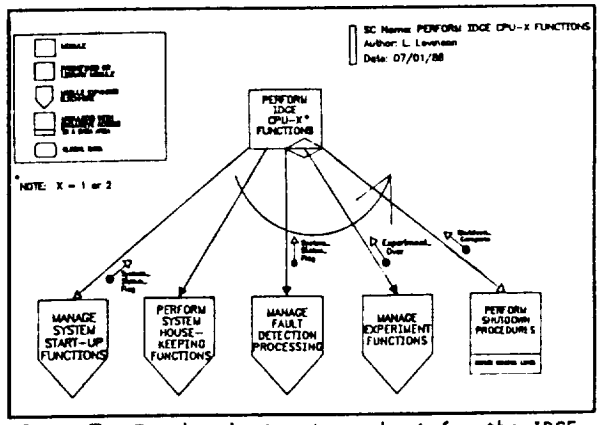

Figure 7. Top level structure chart for the IDGE flight software.

SCs are immensely helpful as a visual aid, both in initial system development and, later, during maintenance. They are an excellent mechanism for obtaining a quick overview of the system structure as it exists at any level of the system.

Module prologues contain a summary of all relevant interface information, as well as a complete change history for the module. They are extremely useful during initial system coding and debugging, particularly if this is being done by someone other than the system designer. They can also be quite helpful to a maintenance programmer, by providing critical information which might otherwise be missed. The module prologue template being used on the IDGE project is shown in Figure 8.

```
*****************************************************************
*                                                               *
* Purpose:                                                      *
*                                                               *
* Input Data:                                                   *
* Output Data:                                                  *
* Module Header:                                                *
*                                                               *
* Argument List:                                                *
*   Argument Name          Type   Use   Description             *
*   -------------          ----   ---   -----------             *
*                                                               *
* Global Variables:                                             *
*   Variable Name          Type   Use   Description             *
*   -------------          ----   ---   -----------             *
*                                                               *
* External References:                                          *
*   Unit Name      Module Name                                  *
*   ---------      -----------                                  *
*                                                               *
* Invoking Routines:                                            *
*   Unit Name      Module Name                                  *
*   ---------      -----------                                  *
*                                                               *
* Development History:                                          *
*     Author          Date      Description of Change          *
*     ------          ----      ---------------------          *
*                               Original version               *
* Notes:                                                        *
*                                                               *
*****************************************************************
```

Figure 8. The module prologue template being used on the IDGE project.

PDLs, also referred to as pseudocode, provide a detailed description of the module logic, using structured English to define that logic. An example is shown in Figure 9. If the PDLs are done properly, coding of the modules should be greatly simplified, as all the logic will have been thought out ahead of time.

In addition to simplifying the coding process, however, PDLs are also very helpful to a maintenance programmer. By providing a textual description of each module, they ease the task of understanding both the individual modules and the system as a whole.

PDLs for the IDGE system are being completed in an iterative fashion with the SCs. An initial set of SCs is generated, which is then used to aid in the initial preparation of the associated PDLs. As these PDLs are written, they then cause a further refinement of the SCs, and so on.

```
{
*****************************************************************
*                                                               *
* pause - perform pause command processing                      *
*                                                               *
*****************************************************************
* Do Case                                                       *
*   Case 1:  Pause_Command_Status_Flag indicates that a Pause Command has
*            been input
*     "Get Current Time"
*     Set Time_to_Resume = Current_Time + Max_Time_to_Pause
*     Set Pause_Command_Status_Flag to PAUSED_WITH_TIMER_SET
*     If Cycle_Phase is greater than or equal to COOLING_WITH_BATH_
*      TEMPERATURE_NOT_STABLE
*     then
*       Set Target_Temperature to Melt_Temperature
*       Set Control_Thermistor to low resolution bath thermistor
*       Set New_Temperature_Requested flag to TRUE
*       If Cycle_Phase is greater than or equal to COOLING_SCN_WITH_BATH_
*        TEMPERATURE_STABLE
*       then
*         Set Stable_Bath_Temperature_Flag to FALSE
*         If Cycle_Phase equals SCN_STABLE_AT_SUPERCOOL_AND_PERFORMING_
*          EXPERIMENT
*         then
*           "Terminate Cycle Events"
*           Set Stable_SCN_Temperature_Flag to FALSE
*         Endif
*       Endif
*       Set Cycle_Phase = START_OF_CYCLE
*     Endif
*   Case 2:  Pause_Command_Status_Flag indicates that an Unpause Command
*            has been input
*     Set Pause_Command_Status_Flag to UNPAUSED
*   Case 3:  Pause_Command_Status_Flag indicates that the experiment is
*            currently paused
*     "Get Current Time"
*     If Current_Time > Time_to_Resume
*     then
*       Set Pause_Command_Status_Flag to UNPAUSED
*     Endif
* Enddo
*                                                               *
*****************************************************************
}
```

Figure 9. PDL for the pause command processing module.

Generation of the IDGE implementation model in the above-described manner is expected to take the full-time effort of one software engineer for approximately two years. At the conclusion of that period, a Software Design Document containing SCs, module prologues, and PDLs for the entire IDGE flight system will be issued.

Development of this implementation model is being guided by information provided in Meilir Page-Jones' "The Practical Guide to Structured Systems Design."[5] This excellent text contains a description of techniques which can be used in making the transition from requirements definition to design, as well as a discussion of the qualities that distinguish a well-designed system from a poorly-designed one.

CODING

IDGE Coding Approach

The IDGE flight software, which will run on an IBM PC-compatible STD bus computer, is being coded in Turbo Pascal. Code optimization, if required, will be done in 8086 assembler, and the Turbo assembler provided with Turbo Pascal used for code assembly. Coding is expected to take approximately nine person-months.

As the project is still in the design phase, no flight code for the system has as yet been written. Software to operate the IDGE engineering unit - engineering code - has, however, been written, and this was done directly from flight system PDLs. While this code is thus based on the flight system design, it was written by someone other than the system designer. This approach was used so as to allow work to continue on the flight software design, while still providing the hardware team with software to operate the experiment. When the actual flight code is written, the coding will be

done by the system designer, as it improves reliability to have the entire system developed from start to finish by a single individual; however, for use in testing engineering hardware, this alternate approach was tried and found to be quite beneficial. The software design work continued uninterrupted and, at the same time, code was written for the portion of the system already designed - at the rate of approximately one module (averaging approximately thirty lines of code) every two to three hours. Since little debugging was required, the software system was operational in a relatively short period of time.

This alternative approach to software development is currently increasing in popularity. However, to be most successful, it is important that all three components of the module model - structure charts, module prologues, and PDLs - be available to the coder. If any one of these three items is missing, the efficiency of the coding process will be diminished.

TESTING

As previously mentioned, two types of testing will be done on the IDGE project: unit testing and integration testing.

Unit Testing

Unit testing of IDGE flight software modules will be accomplished using test matrices. A test matrix is a form on which all possible paths through a unit's code are represented. It is generated by taking every conditional statement in a unit's PDL - i.e., every statement for which the path through the logic will differ based on existing conditions - and listing each distinct condition on the form. The unit is then tested by setting up the initial conditions so as to guarantee that each of these paths is taken at least once. As each path through the code is successfully executed, the associated item on the test matrix is checked off. When all items have been checked, that unit has completed unit testing and is ready to be integrated into the system as a whole.

Integration Testing

Integration testing of the IDGE flight software will begin with a functional test of the system under nominal operating conditions. Several complete mission profiles will be run, during which different types of display data will be monitored. This display data will be used to verify that the system functional requirements, as specified in the Software Functional Requirements document, have been properly implemented. In addition, data taken and stored during experiment operation will be checked following each run in order to confirm proper operation.

Once the unit has been successfully tested under nominal operating conditions, the system's built-in fault tolerance features will be tested. This will be done by deliberately introducing errors (e.g., by disconnecting various components) and then observing the system's response, as indicated by the display data.

Testing of the IDGE flight software in this manner would normally be expected to require the full-time effort of one software engineer for approximately eighteen months. Flight schedule realities, however, may mean that testing will instead need to be completed in three to six months. As a consequence, it may be necessary to increase the project staff during the latter portion of the development effort, in order to accommodate this accelerated schedule.

CONCLUDING REMARKS

The software development approach being used on the IDGE project and described above was selected based on the specific needs of the IDGE project. The approach used is essentially that described by Ward and Mellor in Reference 4, but tailored as appropriate to the circumstances. Tailoring of the approach is, however, not at all contrary to what was intended by Ward and Mellor, as they indicate clearly in the following: "We must emphasize that the models we have laid out above do not constitute a development methodology for a project. Each project must tune, or tailor, this general scheme for its own use."[6]

When future projects are choosing the approach most appropriate to their needs, it is hoped that the IDGE experience, as described in this paper, will be of some assistance.

ACKNOWLEDGEMENTS

The authors would like to thank Michelle Oriold for her invaluable assistance in the preparation of this document.

REFERENCES

1. M. E. Glicksman, E. A. Winsa, R. C. Hahn, T. A. Lograsso, S. Tirmizi, and M. E. Selleck, "Dendritic Solidification Under Microgravity Conditions," AIAA 26th Aerospace Sciences Meeting, Reno, NV, Jan. 1988.

2. E. Winsa, M. Glicksman, G. Kraft, D. Miller, and R. Abramczyk, "Flight Hardware and Tele-Operations Supporting the Isothermal Dendritic Growth Experiment Aboard the Space Shuttle," AIAA 27th Aerospace Sciences Meeting, Reno, NV, Jan. 1989.

3. Tom DeMarco, Structured Analysis and System Specification, Prentice-Hall, Englewood Cliffs, 1979.

4. Paul T. Ward and Stephen J. Mellor, Structured Development for Real-Time Systems, 3 vols., Prentice-Hall, Englewood Cliffs, 1985.

5. Meilir Page-Jones, The Practical Guide to Structured Systems Design, Yourdon Press, New York, 1980.

6. Ward and Mellor, Structured Development for Real-Time Systems, 1:39.

NASA
National Aeronautics and Space Administration

Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA TM-102412 AIAA-90-0744		

4. Title and Subtitle

Flight Software Development for the Isothermal Dendritic Growth Experiment

5. Report Date

6. Performing Organization Code

7. Author(s)

Laurie H. Levinson, Edward A. Winsa, and Martin E. Glicksman

8. Performing Organization Report No.

E-5170

10. Work Unit No.

694-23-03

9. Performing Organization Name and Address

National Aeronautics and Space Administration
Lewis Research Center
Cleveland, Ohio 44135-3191

11. Contract or Grant No.

13. Type of Report and Period Covered

Technical Memorandum

12. Sponsoring Agency Name and Address

National Aeronautics and Space Administration
Washington, D.C. 20546-0001

14. Sponsoring Agency Code

15. Supplementary Notes

Prepared for the 28th Aerospace Sciences Meeting sponsored by the American Institute of Aeronautics and Astronautics, Reno, Nevada, January 8-11, 1990. Laurie H. Levinson and Edward A. Winsa, NASA Lewis Research Center; Martin E. Glicksman, Rensselaer Polytechnic Institute, Troy, New York.

16. Abstract

The Isothermal Dendritic Growth Experiment (IDGE) is a microgravity materials science experiment scheduled to fly in the cargo bay of the shuttle on the United States Microgravity Payload (USMP) carrier. The experiment will be operated by real-time control software which will not only monitor and control onboard experiment hardware, but will also communicate, via downlink data and uplink commands, with the Payload Operations Control Center (POCC) at NASA George C. Marshall Space Flight Center (MSFC). The software development approach being used to implement this system, which will be the focus of this paper, began with software functional requirements specification. This was accomplished using the Yourdon/DeMarco methodology as supplemented by the Ward/Mellor real-time extensions. The requirements specification in combination with software prototyping was then used to generate a detailed design consisting of structure charts, module prologues, and Program Design Language (PDL) specifications. This detailed design will next be used to code the software, followed finally by testing against the functional requirements. The result will be a modular real-time control software system with traceability through every phase of the development process.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
Computer systems design; Software development; Software engineering; Real-time operation; Real-time control; Flight software; Process control; In-flight monitoring; Computer techniques; Automatic control; Embedded computer systems; Data flow analysis	Unclassified – Unlimited Subject Category 62

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No of pages	22. Price*
Unclassified	Unclassified	10	A03

NASA FORM 1626 OCT 86

*For sale by the National Technical Information Service, Springfield, Virginia 22161