RAPID PROTOTYPING VIA AUTOMATIC SOFTWARE CODE GENERATION FROM FORMAL SPECIFICATIONS: A CASE STUDY

Dr. S. Rahmani
A.G. Stone
W.S. Luk
S.M. Sweet

Rockwell International Corporation
12214 Lakewood Boulevard,
Downey, CA 90241
(213) 922-2478

Abstract

This paper describes an approach for defining system and software requirements and validating them through rapid prototyping. It provides the results of a case study for generating prototype software (representing the total system hardware and software) directly and automatically from the requirements. The paper addresses system/software definitions, which include the requirements and design (architecture). The approach consists of modeling and prototyping. The case study described here applied this approach to selected systems of Boeing 747-400 aircraft. A formal model of system specification was generated. The rapid prototyping task automatically generated thousands of Ada source lines of code from the specification model. The tool used on the project was Statemate. The software was executed successfully the first time. The functions and behavior of the system were demonstrated and validated by its users. This study indicated that early execution and validation of system requirements, through the use of formal modeling and rapid prototyping with direct user involvement, can be accomplished.

Introduction

The complexity of avionics is increasing in step with the increased processor power available. The trend is to migrate more decision making and intelligence into the avionics. For example, 10 years ago, commercial aircraft were manufactured for a flight crew that consisted of three people: the pilot, the copilot, and the flight engineer. Today, aircraft are manufactured for a two-person flight crew, the flight engineer has been replaced by an avionics system that performs those duties. In addition, today's aircraft have systems that didn't even exist 10 years ago: the onboard maintenance systems (OMS), a distributed and integrated system for detection and isolation of the faults throughout the system[1]; satellite communications system; traffic alert and collision avoidance system; and others.

Avionics systems interfaces are becoming much more complex as more data are being shared between them. Ten years ago, communication between systems was all point to point or broadcast. This included digital, analog, and discrete data. Today, there is an increasing use of high speed local area networks, and most sensor data is converted to digital format at the sensor before being transmitted. This makes almost all data accessible to all avionics systems, resulting in more complex integration.

Development methods that worked well for simpler, less complex systems have not scaled up well. Requirements errors are finding their way into the final product. More emphasis is being put on generating the right requirements before detailed design and development occur. There are several computer-based tools available that provide formal definition, analysis, simulation, prototyping, and other capabilities that are utilized to define and evaluate requirements. They are commonly referred to as computer-aided systems design (CASD) tools.

Given the above challenges, this paper describes the results of a case study on defining and applying a process for specification and validation of the requirements. First, it identifies the objectives established for the study. Next, it describes the approach for achieving the objectives including (1) an overall description of the case study, (2) the process used for formal definition of the system requirements, (3) description of the formal model itself, (4) description of the validation and rapid prototyping process, (5) rapid prototyping results, (6) description of the tools used throughout this approach, and (7) some lessons learned from this case study. The last section of the paper includes the summary and conclusions.

Objectives of Case Study

The following objectives were identified for the case study.

1. Systems Engineering Process—At the time of the study there was no formal systems engineering process at Collins Air Transport Division (CATD) of Rockwell International, and no clear path for downflow of systems requirements to the software and hardware engineering disciplines existed. There was also skepticism about the viability of the approach in the areas of modeling, simulation, and requirements validation.[2,3,4] A primary objective was to assess some of the basic concepts before they were incorporated into a process and applied to product development.

2. Tool Evaluation—A new class of CASD tools were available. An objective was to select one of the more capable ones and evaluate it under simulated project conditions. The main areas of consideration were (1) effective use of simulation and prototyping for system evaluation and validation, (2) integrated project data base organization and interfaces, (3) configuration management and control, and (4) document generation.

3. Systems Engineering Expertise—This objective was to develop individuals with systems engineering skills. The primary focus of the case study was to develop techniques for defining, evaluating and testing, and documenting system requirements effectively before they are supplied to the build organizations like software and hardware engineering.

4. Fault Isolation—An assessment was provided on how to improve fault isolation capability for air transport category aircraft.

5. Rapid Prototyping—A determination was made as to whether it was possible and reasonable to develop rapid prototypes directly from a requirements model. Ways were identified to use the rapid prototype.

Case Study Description

While the objectives remained of a general nature, the task was specific: formally define the system and software requirements (for fault detection and isolation) of Boeing's 747-400 OMS, then validate them through rapid prototyping. At the time, several CASD tools were available on the market. Statemate[5], a CASD tool set, was selected on the basis of its overall capabilities.

Although primarily interested in developing system engineering expertise, the team was also interested in the assessment and improvement of fault isolation capabilities for air transport category aircraft, which led to the selection of the existing Boeing 747-400 OMS as the target system. There were several other reason for this selection:

1. CATD had developed both the system hardware and software of the central maintenance computer (CMC), the heart of the OMS where fault logic equations are managed and executed; therefore, it was a known reference point.

2. Since the program was still active, expertise was accessible.

3. The OMS provided a centralized aircraft fault isolation capabilities, thus it interfaced with almost all aircraft systems.

4. The user interface to the CMC was through a control and display unit (CDU), which could be graphically prototyped on a work station.

5. Development of this system was labor intensive, especially the part for the fault logic equations. Any automation or improvement in fault logic creation would improve productivity.

6. Since the system already existed, reverse engineer techniques could be used to model the system (this was easier than creating a model for a brand new product.)

The team itself consisted of four individuals, each bringing to it a unique background and a broad range of expertise. Due to the limited amount of resources available, only a limited number of Boeing's OMS interfacing systems could be included in the study. Thus, the definition process began with defining the scope of the target system.

Model Definition Process

As outlined in the case study objectives, there was no formal systems engineering process at CATD, and no clear path for the downflow of systems requirements to the software and hardware engineering disciplines; thus a formalized process definition for the case study had not been established. The approach for the case study, as shown in Figure 1, was as follows:

1. Determine the scope of the model to be constructed.
2. Develop modeling guidelines.
3. Develop a formal definition model using reverse engineering techniques.
4. Validate subsystems models using simulation.
5. Integrate the subsystems into the system model and validate the entire system using simulation.
6. Using Graphics Kernel System (GKS) as a graphics engine, develop a graphic representation of the user interface.
7. Develop prototype Ada code of the system and link it to the graphic model of the user interface.
8. Validate the system specifications using the prototype.

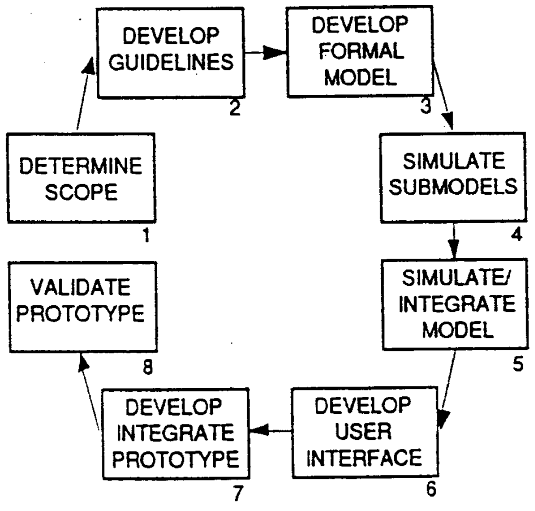


Figure 1. Model Definition and Validation Process

The scope of the model was based on the limited amount of resources and personal expertise of the team members. Previous to this assignment, the team had spent a considerable amount of time studying the fault detection and isolation capabilities of various Boeing 767 aircraft systems. As a result of this analysis the following two systems were selected for modeling: (1) the electrical power generation and distribution (EPGD) system, which is a complex system with limited built-in-test-equipment (BITE) capability, and (2) the communication system (COMM), which is less complex and has good BITE capability. In addition, formal models were developed for both the CMC, which executes the fault logic equations, and CDU (for user interface).

Modeling guidelines were established to ensure that the work of each team member interfaced with those of the others. Since the modeling tool primarily provided a formal modeling language, specific modeling techniques were left to the individual team members to define and use.

Test and evaluation of the details of the definition model was performed by means of simulation, a technique in which the user can interactively or dynamically walk step-by-step through the model to ensure that it is indeed the correct conceptual view of the system. In parallel to the modeling activity, a graphic engine was used to develop a graphic model of the user interface, the CDU.

Specification Model

Although the developed specification model was only a subset of Boeing's total fault detection and isolation system for 747-400, the model created was considered to be very large for a research and development project. Upon completion, it contained approximately 55 functional and data flow diagrams (activity charts) and 200 state transition diagrams (state charts), and took approximately 6 months (2 man years) to develop. Because of the model size, a major part of the above time was spent on integration, test, and validation of the model. In fact, the completed target system was never prototyped in its entirety because the model was too big for the DEC Ada compiler to compile. As a result, prototypes were developed per single interfacing system. For example, the communication system prototype included the CDU, the communication fault logic equations of the CMC, and the fault isolation characteristics of the communication system.

The EPGD model was the largest of the four subsystems modeled. By itself it contained approximately 15 activity charts and 55 state charts. In terms of computer memory, the electrical system was about 8 MBytes in size. The EPGD model, as shown in Figure 2, included 37 line replaceable units (LRU), data and power interfaces (e.g., buses) and the associated internal BITE capabilities of several LRUs. LRUs included in the EPGD system model were four generator control units (GCU), four generator circuit breakers (GCBs), four bus tie breakers (BTBs), four ac power buses, four power distribution units (PDUs), two auxiliary power units (APUs), two APU generator control units (AGCUs), two bus control units (BCUs), two external power connectors (XPC), two synchronous buses (SYNC Bus) and one split system breaker (SSB). An example of an activity chart, showing the functionality and flows (both electrical power and information) for an integrated drive generator system, is shown in Figure 3. The block with rounded corners represents the behavior of these functions (modeled separately as a state transition diagram).

The COMM model, on the other hand, was quite smaller. It included 16 LRUs, including two HF receivers, three VHF receivers, the associated couplers and antenna, and three radio control panels (RCPs). Data and power interfaces were also included in the system model along with the associated internal BITE capabilities of specific LRUs.

In developing the conceptual model, each interfacing system was modeled from two distinct viewpoints: functional and behavioral. In Statemate, this model was described through the use of activity charts and state charts. While activity charts described the associated system functions, state charts addressed the corresponding behavioral aspects of these functions. In the system model, for example, activity charts were used to identify the BITE capabilities of "smart" LRUs (those LRUs that were capable of detecting and reporting their internal faults) while the internal behavior (fault processing) was described through state charts.

While the distinction between function and behavior remained constant throughout the model, each subsystem employed different modeling techniques. The COMM, for example, developed a technique for fault propagation, a step-by-step process of visually observing fault traveling throughout the COMM to the CMC. The electrical system, on the other hand, developed techniques for modeling bidirectional data as seen within the power buses. Timing effects of faults, such as delays and status history, were also incorporated in the electrical system model.

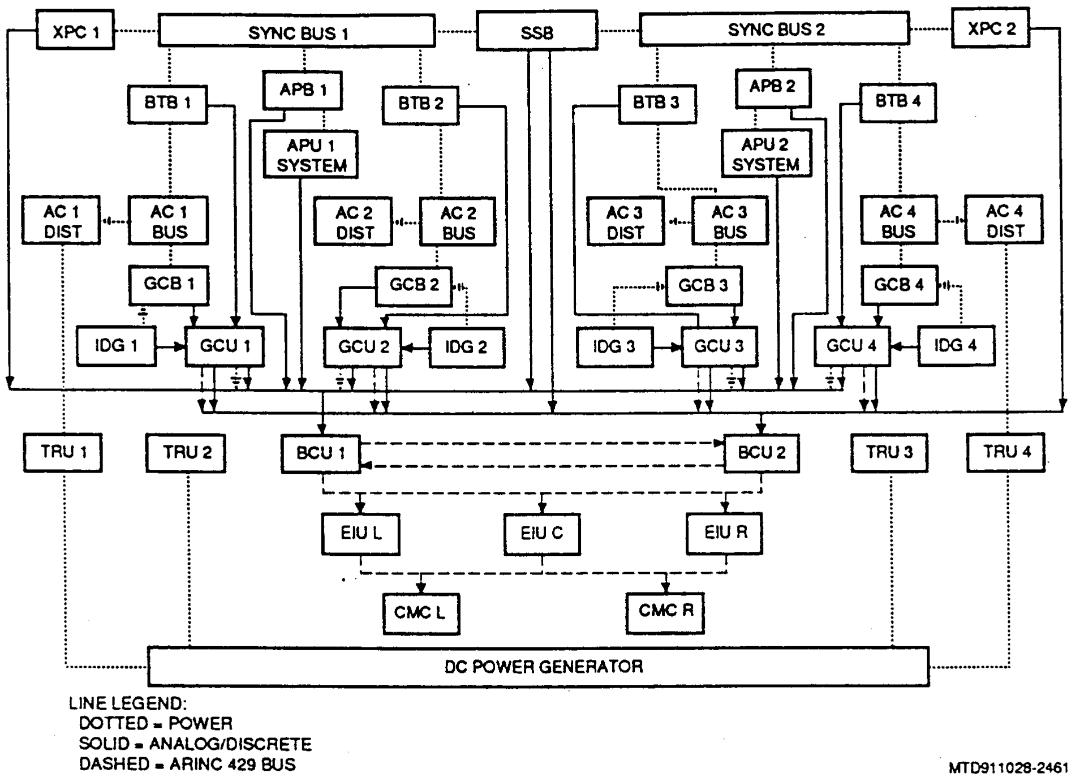
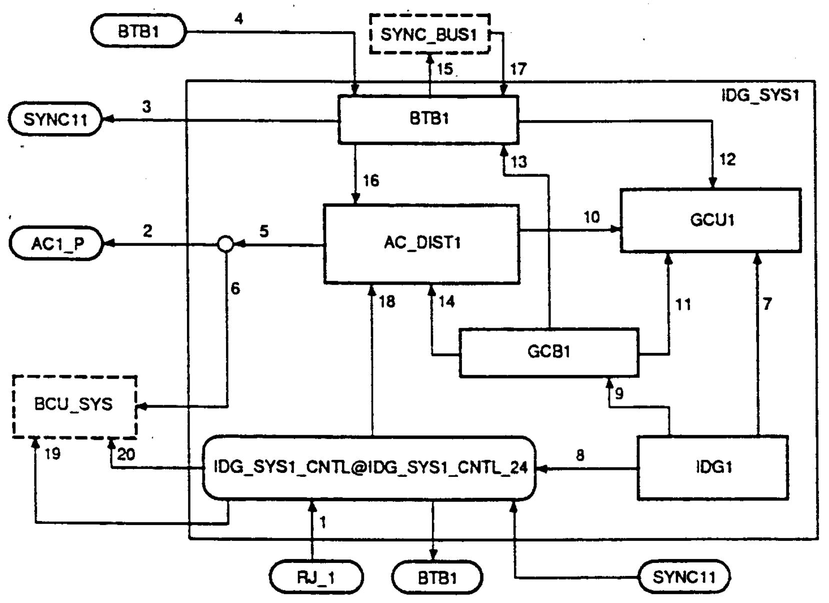


Figure 2. Block Diagram of EPGD (AC Portion)



1	RJ_GCU1_M
2	AC_DIST1_PWR
3	SYNC_BUS1_PWR
4	BTB1_SYNC1_PWR
5	AC_DIST1_PWR
6	AC_DIST1_PWR
7	IDG1_S
8	IDG1_GCB1_PWR
9	IDG1_GCB1_PWR
10	AC_DIST1_PWR
11	GCB1_S
12	BTB1_S
13	GCB1_AC_BUS1_PWR
14	GCB1_AC_BUS1_PWR
15	BTB1_SYNC1_PWR
16	BTB1_AC_BUS1_PWR
17	SYNC_BUS1_PWR
18	AC_DIST1_PWR
19	GCU1_BCU1_TS
20	GCU1_BCU2_TS

MTD 911104-2481

Figure 3. Example of an Activity Chart, Showing the Functionality of the Integrated Drive Generator System

Rapid Prototyping Process

Along with the overall project objectives, a number of more specific objectives were established for validation of the system. First, the validation was to involve both the customer and developer (contractor). Since the customer might not be familiar with the tools and modeling details, the system model was to be validated in an operational and user-friendly environment. Second, validation was to be done early and at low cost. This would eliminate any options dealing with extensive development and test of software or hardware. Rapid prototyping through the use of automatic code generation was key to achieve this objective. Third, it was to be done thoroughly to allow both white-box and black-box testing of the system model. Finally, the validation procedures were to be reusable during both the early validation and postbuild validation of the system.

The validation approach, shown in Figure 4, met the above objectives. Use of the Statemate tool set provided the capability to generate executable version of the system model in Ada for validation (directly from the specification). This prototype code represented the total and integrated system. It included software simulating the hardware components and software for the anticipated software components. White-box testing was done through simulation and black-box testing of the system was provided by prototyping. Both simulation and prototyping represented real-time execution of the requirements model within a simulated time environment. Once the system model was available, its validation could be done very cost effectively within a few hours through the use of automatic Ada code generation capability (i.e., without any design or implementation of software or hardware.)

Through the use of a cockpit type user interface, represented as a CDU panel, the prototype code was executed in a user-friendly and operational type environment. System architecture diagrams, along with fault menus associated with each LRU, were used for selecting various operational scenarios. The model allowed as many simultaneous system environment inputs as desirable. Upon execution of each set of input scenarios, the system prototype functioned accordingly and displayed the results on the CDU panel.

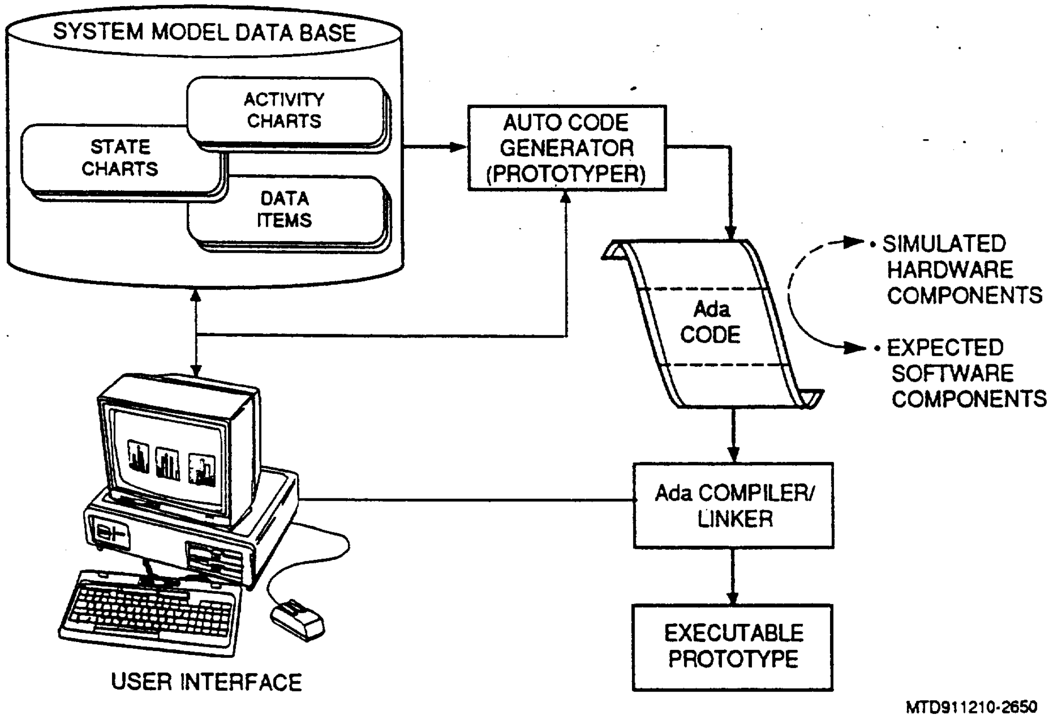


Figure 4. Validation Approach Using the Statemate Prototype

Certain modifications to the generated prototype code were necessary for integration with the GKS graphics. Since the prototype code was very structured and cryptic, the prototyping tool created two source files specially for integrating custom routines. These files provided "hooks" to detect and set changes in the model, and places to insert developer's source code to be executed at different stages (such as the beginning and at the end) of the prototype.

Much of the debugging of the integrated prototype was done to resolve GKS execution ambiguities. The automatically generated source code, even though large, was "bug-free". The biggest debugging effort was to "dis-synchronize" execution of GKS routines and Statemate prototype. To realistically reflect the operating environment, the GKS input function and the prototype functions were created as separate processes. That is, the system operator's actions were independent of the prototype model. However, the GKS input function used a polling mechanism and stopped executing the other programs (the prototype) during any input request. Separate system level processes were created at the simulation platform (VAX VMS). One process was used for running the prototype and another for GKS graphics. Communications between these two process were done through system level interprocess mail functions (VAX VMS Mailbox).

Tools

Automation was a key element of this approach. The goal was to maximize use of tools available from vendors and minimize those developed in house. Special consideration was given to integrating the tools and their data bases.

Several candidate systems engineering and design tools were investigated for this approach including ADAS, Foresight, RDD-100, and Statemate. Several criteria were identified and used for evaluation of these tools such as (1) capability to model system functional, behavioral, architectural, and information viewpoints, (2) capability to perform static (syntax) tests and simulation, (3) automatic generation of software code from model, executable on host platform for requirements validation by the user, (4) capability to generate, test and, execute models for large systems with adequate performance, and (5) support of multiple simultaneous users potentially located in geographically dispersed facilities. Computer-aided software engineering (CASE) tools such as Teamwork and Software Through Pictures were not included since they did not meet the main selection criteria.

The above evaluation identified Statemate as the preferred tool among the available choices. It is a systems engineering tool mainly used during the "front-end" part of system development. Based on the traditional "water fall" model, this tool covers formal definition of the system requirements and design (including allocation of functions to hardware and software), and validation of the requirements and design through execution of the formal requirements and design models using simulation and rapid prototyping (through automatic code generation).

As an example, Figure 5 illustrates how the project data base was used to capture the systems requirements. The data base included three views of the system requirements modeled during this study (functional, behavioral, and information/data), test procedure and report files, GKS graphics and text panels, and templates for automatic document generation. The requirements were validated through performing the activities shown on the right side of the figure.

As represented in Figure 5, Statemate consists of a number of integrated software tools: (1) Kernel, which includes graphics editors (used for generating the definition model), static (basic) tests tool, data base query utility, and standard report generator; (2) Analyzer that handles dynamic execution and simulation of the models; (3) Prototyper that allows rapid prototyping of the formal models in Ada or C; and (4) Documentor that can be used to generate user-defined templates for documentation (e.g., DOD-STD-2167A). This tool set creates and uses an integrated data base.

A key and rather unique feature of Statemate is its capability to perform dynamic testing and simulation of the model. It focuses on the sequence of the occurrence and execution of various elements in the model including (1) external events (e.g., control commands, timing events, interrupts, etc.), (2) internal conditions (failures, time-outs, reconfiguration conditions, etc.), and (3) functional elements. This capability is especially important for specification of systems with large and complex control characteristics (e.g., modes, events, and time-critical functions). The simulation (and prototyping) capability of the tool provides real-time execution of the system model within a simulated time environment.

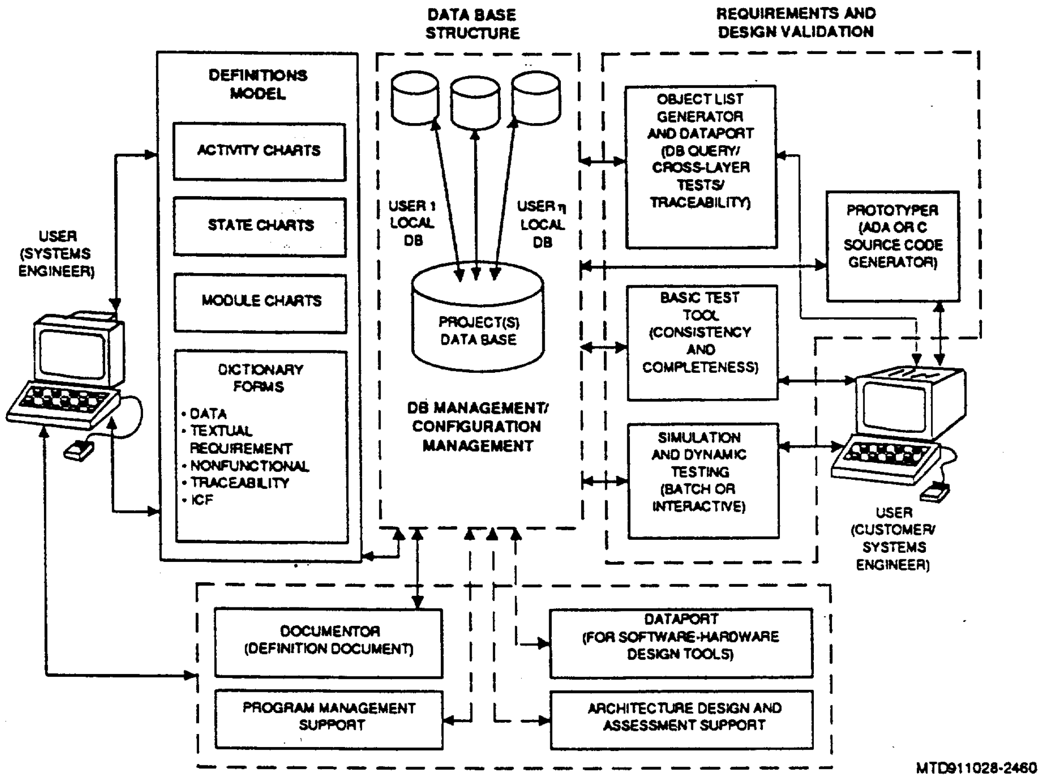

DATA BASE STRUCTURE
REQUIREMENTS AND DESIGN VALIDATION
DEFINITIONS MODEL
ACTIVITY CHARTS
STATE CHARTS
MODULE CHARTS
DICTIONARY FORMS
• DATA
• TEXTUAL REQUIREMENT
• NONFUNCTIONAL
• TRACEABILITY
• ICF
USER (SYSTEMS ENGINEER)
USER 1 LOCAL DB
USER n LOCAL DB
PROJECT(S) DATA BASE
DB MANAGEMENT/ CONFIGURATION MANAGEMENT
OBJECT LIST GENERATOR AND DATAPORT (DB QUERY/ CROSS-LAYER TESTS/ TRACEABILITY)
PROTOTYPER (ADA OR C SOURCE CODE GENERATOR)
BASIC TEST TOOL (CONSISTENCY AND COMPLETENESS)
SIMULATION AND DYNAMIC TESTING (BATCH OR INTERACTIVE)
USER (CUSTOMER/ SYSTEMS ENGINEER)
DOCUMENTOR (DEFINITION DOCUMENT)
PROGRAM MANAGEMENT SUPPORT
DATAPORT (FOR SOFTWARE-HARDWARE DESIGN TOOLS)
ARCHITECTURE DESIGN AND ASSESSMENT SUPPORT
MTD911028-2460

Figure 5. Elements of System Modeling and Prototyping Method/Tool (Statemate-Based)

The tool has built-in support functions for configuration management and traceability. Its relational data base (in ASCII format) represents an integrated project data base that can be partitioned into a number or local data bases for multiple users, who may be located remotely from each other, each working on a section of the problem under study. The tool provides adequate control for file access and security considerations.

Capabilities of Statemate were complemented by using another tool to provide user interface for rapid prototyping. The GKS graphics library package was used for this purpose. It provided the graphics engine that linked the user and operational interface capability to the system model.

The tool can generate either Ada or C prototypes. The prototype was compiled and linked with predefined libraries of routines (provided by the tool vendor, i-Logix, as part of the prototyping tool set) to create an executable image for the host computer. Since Ada or C code was generated and i-Logix provided a number of platform dependent software packages, the prototype could be executed on most of the major work stations (such as DEC, Sun, Apollo, and IBM).

The user interface to simulate the operating environment and interact with the prototype could be created either through the Statemate's Panel Graphics Editor (PGE) or other commercial user interface packages such as GKS.

The PGE could define simple graphic representations (buttons, knobs, dials, etc.) of the operational environment. However, the capabilities provided by PGE were inadequate, and the CDU display panel and the system architecture diagrams were created using DEC GKS. These graphic panels provided realistic operational interfaces that were familiar to the customers. All GKS graphics were placed in Ada packages and incorporated into the prototype code generated by Statemate.

Each graphic button on the GKS CDU panel represented a signal (or event). When GKS detected a button was selected or "pressed" the GKS routines set a corresponding event and triggered certain behavior in the model. Faults injected into the model through the system architectural diagrams were represented as conditions. GKS routines were also added at the beginning and at the end of prototype execution to start and stop the GKS graphic kernel.

Rapid Prototyping Results

The major accomplishment in this study was that the prototype of the specification model in Ada, generated automatically and executed without errors the first time after it was successfully compiled and linked. The behavior of the prototype truly reflected that of the model. That is, every facet of the model behaved as expected. In fact, the prototype behaved so realistically that it even illustrated certain unexpected behavior of the actual OMS system. Every time an aircraft is powered up, the electrical power transient causes the OMS to record "nuisance" faults from various aircraft systems. When the electrical power (as a condition) was put into the model, the CMC portion of the model actually exhibited behavior similar to the real OMS and detected nuisance faults from the COMM.

Although a major part of this accomplishment was a result of the modeling, validation, and automatic code generation capabilities provided by the tool set, several other factors played important roles throughout this effort. Among these factors were (1) partitioning of the specification model in modular elements such that each member of the systems engineering team could work on a separate portion of the model, (2) extensive use of state transition diagrams for modeling the fault logic equations graphically, (3) definition of relatively consistent terminologies and notations for representing various components of each system (e.g., buses and power lines), (4) selection and proper use of the GKS for implementation of the system user interface and integration with the Statemate model, and (5) use of a "tool independent" systems engineering process that employed Statemate (or potentially other similar CASD tools) for its activities as opposed to using a process that was entirely based on the tool.

The prototype was shown to Rockwell's engineers who worked on the actual OMS system. The only complaint that they had was that minor details weren't put into the model. The prototype was then demonstrated to the OMS group at Boeing. The reception was both astonishment and excitement. Boeing engineers were very surprised that the code automatically generated from a requirement model could behave so close to the real system.

Both Boeing and Rockwell recognized that potential impacts of such a rapid prototype. First, requirements could be validated by the true end user—someone who might not have the engineering background to understand a formal requirements model. This could eliminate many ambiguities between developers and end users. Second, the prototype was generated with no software development effort other than the GKS user interface development. All requirement changes were done on the model, and a new prototype could be generated in a matter of hours.

The primary control mechanism of the prototype was a clock counter. This allowed real-time execution of the model within a simulated time environment. For each internal clock tick, the prototype executed all transitions (regardless of the number) within that clock period. This could be a misrepresentation of a real-time system. For instance, the processor in a real-time system has definite constraint on the number of instructions it could execute within a given time period. The prototype could not handle this detailed requirement, and the system engineer building the model had to ensure that the number of instructions (low level) or state transitions within a clock cycle were realistically comparable to the processors speed in the final system.

For the model of the Boeing 747-400, including the CMC, COMM, and EPGD, the prototype code was extremely large (about 140,000 source lines of Ada code for COMM, and the associated portion of CMC and CDU systems only). It was estimated that the size (in terms of number of source lines of code) was roughly 20 to 25 times that of the hand-crafted code.

The sizing problem was principally caused by the large symbol table created by the tool. A symbol table was generated to define all elements within the model. However, the tool generated 5 to 10 Ada (or C) variables for each element of the model (state, activity, data, event, condition, etc.). The model included between 4 to 5 thousands of such elements. Together with other internal control elements generated by the Prototyper tool, the symbol table file itself was close to 40 thousand lines of Ada code! However, the tool vendor has indicated that the next release of the tool would include a much more optimized code translator, and the symbol table would be eliminated.

The size had caused problems in compiling and referencing the symbol table (for hooking up the user interface routines). The upper limit of the DEC Ada compiler was reached during this effort, and it became necessary to manually break up the symbol table file into multiple files for compilation. Eventually, the internal limits of the DEC Ada compiler were encountered. At that point, the team was unable to integrate the full electrical system as part of our overall prototype.

The problem associated with the size of the model and prototype was resolved by creating submodels for each major part of the system (COMM and EPGD). Each submodel included simulated interfaces representing the others. The current optimization effort by the tool vendor is expected to provided a more complete resolution for this problem.

Lessons Learned

The basic concepts for modeling, simulation, and prototyping applied to the case study were sound. They could provide the basis of a formal systems engineering process. There are several ways to provide requirements to downstream users (such as hardware and software), which include requirements documents, models, prototypes, and direct porting of requirements from the CASD tool to CASE tools (this requirement was not actually attempted). Requirements validation was not only possible, but proved to be an extremely powerful approach to assessing whether the requirements were correct. Requirements prototypes were developed in a relatively straight forward fashion and provided significant insight.

The tools used for this case study provided many capabilities useful to systems engineers that were not available from CASE tools. The main tool, Statemate, was one of the most capable tools of its kind, but was still relatively immature (the CASD industry is relatively young). It was observed which formal specification languages were large and syntactically rich; however, they took a while to learn. The simulation and prototype capabilities provided opportunity to evaluate behavioral aspects of the system. An integrated project data base, that worked well on multiple types of platforms, accommodated multiple simultaneous users. It also allowed geographically remote users to have access to the data base. Configuration management capability was provided as part of the tool and found to be very crucial. Automatic document generation from the requirements data base could be provided to save document generation time and ensure compatibility of the document with the validated model.

Individuals working on the case study developed some of the fundamental systems engineering skills for defining and evaluating system requirements. Experience from the case study was leveraged to form the basis of a formal systems engineering process.

As stated previously, a working rapid prototype was created from the requirements model on the first attempt. It proved to be extremely useful for demonstrating realistic system operations and functions without burdening the user with the formal language of the tool.

The size of the model and prototype, combined with the limitation of the tools in handling very large models, created a major issue during this case study. An important lesson learned was that a separate model should be created for each system being modeled (CMC, COMM, and EPGD). In fact, the DEC work station (VAXStation 3100 with 24 Mbytes of memory) performance suffered when the model became very large. There were several important reasons for breaking up the complex systems into multiple models:

1. The load would be eased on the computer(s) running the simulation. As the model got more complex, the size of the prototype grew in orders of magnitude. Eventually, the system was overloaded, unable to compile the prototype code generated.

2. Each prototype of a system (or subsystems) could be run on separate computers. This would make the user interface easier to manage.

3. The most important benefit of breaking up the model was modularity. In reality, not all avionics systems inside an aircraft are packaged into one system, but are integrated through well-defined interfaces. By breaking up the model, the systems engineer would confine the domain of a system (through a model). This would make debugging and modification of the model much simpler.

Conclusions

This case study demonstrated that through the use of formal specification methods and tools, rapid prototyping (through automatic code generation) can be used for validation of system requirements and design by the system users. This early validation would identify the errors associated with system requirements and design, and prevent their propagation. The study indicated that while automatic code generation could be very effective in generating a system prototype, the front-end formal modeling effort and its attributes (such as capturing and integrating multiple views of the system) play an important role in adequate validation of the system requirements and design prior to the system implementation.

The study demonstrated that Statemate is a useful tool for formal system modeling and validation through prototyping and automatics code generation. In fact, the survey of the available tools indicated that this tool was one of the few that provided automatic code generation capabilities. However, several features of the tool (Release 3.0) should be improved. One such feature was the size of the automatically generated code, about 20-25 times larger (and less efficient) than the hand-generated code. Another desired feature was the reuse of the components of the formal model (in a more object-oriented fashion). The future release of the tool is expected to improve these features. Some preliminary benchmarks have indicated that the automatically generated code is now about 5 times larger than the hand-crafted one and the code is more readable.

The approach described in this paper is suitable for control-oriented applications where the behavior of the system, based on a set of events and conditions, is to be defined and analyzed. On the other hand, the approach is not very strong in handling data-driven applications. This deficiency can be overcome by adding more object-oriented capabilities to the methods and tools used under the approach. A potential area of future research related to this work includes improvement and validation of the automatic code generators for producin production-quality software directly from formal requirements and design specifications. This would significantly reduce the cost of software engineering process.

Acknowledgment

The authors would like to express their acknowledgment to Jim Curren, Scott Sweet, and Tycho Hayashibara for their support and contributions.

References

1. *Onboard Maintenance System*, Aeronautical Radio Inc., ARINC 624 (1988).

2. Page-Jones, M. *The Practical Guide to Structured Systems Design*. Yourdon Press (1980).

3. *Structured Analysis for Real-Time Systems*. Yourdon Inc. (1984).

4. Hatley, D.J. *The Use of Structured Methods in the Development of Large Software-Based Avionics Systems.* Proceedings of Structured Analysis Workshop (1985).

5. *The Language of Statemate*, i-Logix Inc. Burlington, Massachusetts (1990).