Using the Coats-Mellon Operational Specification

By Mark Coats, Mark McCloskey, and Theo Molla, June 01, 1999
20-26 minutes

The authors are software engineers for Motorola. Mark Coats can be contacted at p26728@email.mot.com. Terry Mellon, CMOS coauthor, can be contacted at mellon@seex.com.

The article "Constructing Operational Specifications," by Mark Coats and Terry Mellon (*DDJ*, June 1995), introduced the Coats-Mellon Operational Specification (CMOS), a methodology for defining user-based scenarios that represent a complete and accurate model of system behavior. For the most part, the article focused on the methodology's constructs and usage, using an automated-teller machine example to show how the method works, producing diagrams that could be transitioned to an object-oriented and/or structured-analysis model. In this article, we'll describe how CMOS has been used since then on real-world projects.

The Mayer Receiver Project, for instance, is a hardware/software development project at the Motorola Space Systems and Services Division (SSSD). The purpose of the project is to provide secure communications. The Mayer receiver is a specialized processor of message packets in a communications system. This small project currently includes 21,000 lines of integrated C++ and Java code and is being updated to add another 10,000 lines. The software controls the hardware to allow data input, then processes that data for dissemination. The software is controlled by a Graphical User Interface (GUI). The CMOS method was used to synthesize, analyze, and validate the software requirements for the receiver.

The Operational Specification

Before discussing the impacts of the method during the development of the Mayer receiver, we'll provide a quick overview of CMOS. Refer to the original article for more details.

The operational specification is a set of diagrams that specify incoming stimuli via actor events, and a system's response to these stimuli. An actor event is an occurrence initiated by an actor at some point in time. The operational specification consists of diagrams that divide behavior into a set of actor events and system responses to those events. The diagrams produced by the operational specification are pure analysis models. They address analysis-phase, system-level behavior only and do not specify data, design, or implementation information.

The CMOS Diagrams

The CMOS method produces three diagrams:

- The actor diagram.
- The actor-event diagram.
- The event-response diagram.

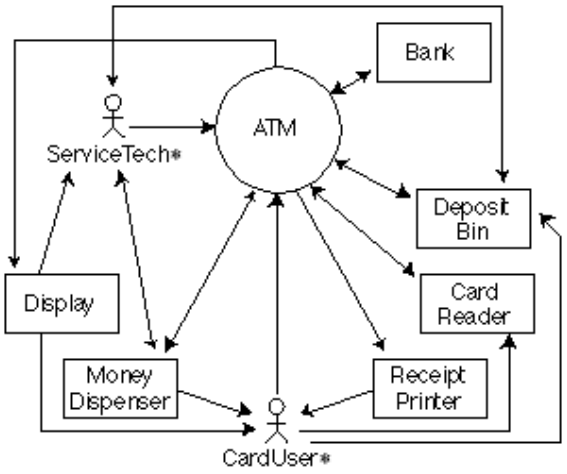

Figure 1

The actor diagram (Figure 1) shows actors that initiate and/or receive events. Actors can be human or nonhuman entities. Each actor event is plotted on the actor-event diagram.

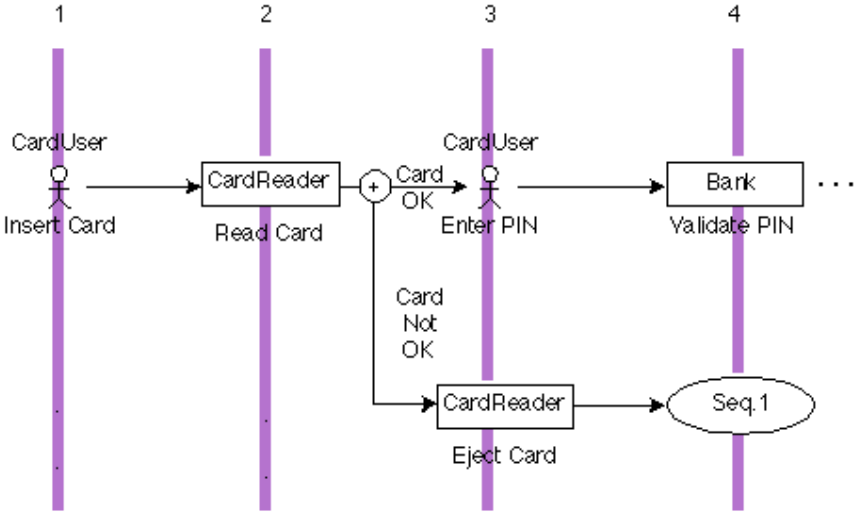

Figure 2

The actor-event diagram (Figure 2) shows how actor's events are related to each other in time. It is similar to UML sequence diagrams with more detail. System scenarios can easily be extracted from this document. Scenarios extracted from this diagram are considered system-level scenarios because each event is caused by a system-level actor action. Each actor event may have a set of system responses (we call them "response bubbles") defined by event-response diagrams (Figure 3).

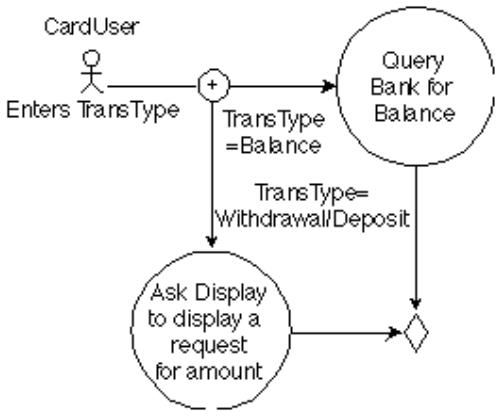

Figure 3

Scenarios extracted from this diagram are normally considered software-level scenarios because the responses to actor events are usually implemented with software. These three diagrams work together to produce a complete specification of the scenarios that define a system's behavior, both at the system and lower levels.

Validating behavior represented in an operational specification involves tracing scenarios through the diagrams starting with an actor event and continuing through the event's corresponding system responses. The ease with which this validation process can be performed lets systems engineers, software engineers, domain experts, and customers help validate system requirements early in the development process. The scenarios are also excellent tools for the development of system-integration test cases that can be inserted directly into a test procedure document or plan.

The Mayer CMOS Process

The Mayer development team was initially given a sketchy set of system requirements for the project. From that, the team spent about eight weeks analyzing the system and its operation to produce CMOS diagrams. A textual requirements document was constructed while creating the diagrams to convey requirements that were not behavioral in nature. A GUI drawing was also constructed to help represent graphical components mentioned by textual descriptions in the diagrams. (This was not required by the CMOS method but is recommended for software that uses a large GUI.) After completion, the CMOS diagrams were used to produce a test-specification document. Additionally, the team estimated the development effort for each response bubble in hours. That data was used as a tool to monitor development efforts and report status.

Mayer Actor Diagrams

The actor diagram was used to determine sources of events for the Mayer receiver system. It was the first CMOS diagram created and helped define the actors of the Mayer software. Besides defining the software actors, the diagram helped to discover which users would initiate events, send and receive events, or only receive events. Figure 4 is an example of this diagram. Once defined, the diagram was frequently used as a reference throughout the rest of the CMOS modeling process.

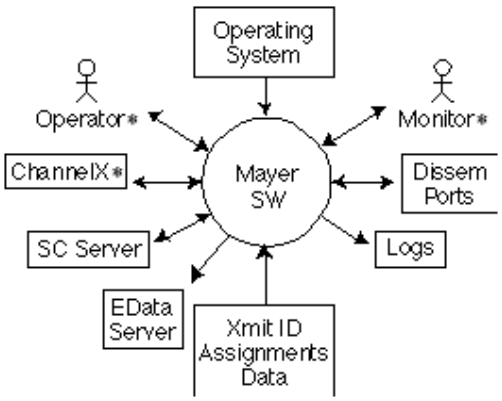

Figure 4

Mayer Actor-Event Diagrams

CMOS actor events are recorded in sequence on actor-event diagrams. Actor events are used to define system behavior in greater detail. Actor-event diagrams are similar to use case sequence diagrams but provide more detail. Figure 5 is a sample of actor-event diagrams for the Mayer receiver system. The diagram reads from left to right across a series of sequences. The left-most event on sequence one starts a scenario. Many scenarios can branch off of a single event. Scenarios proceed to a final sequence on the right, then repeat back to other sequences.

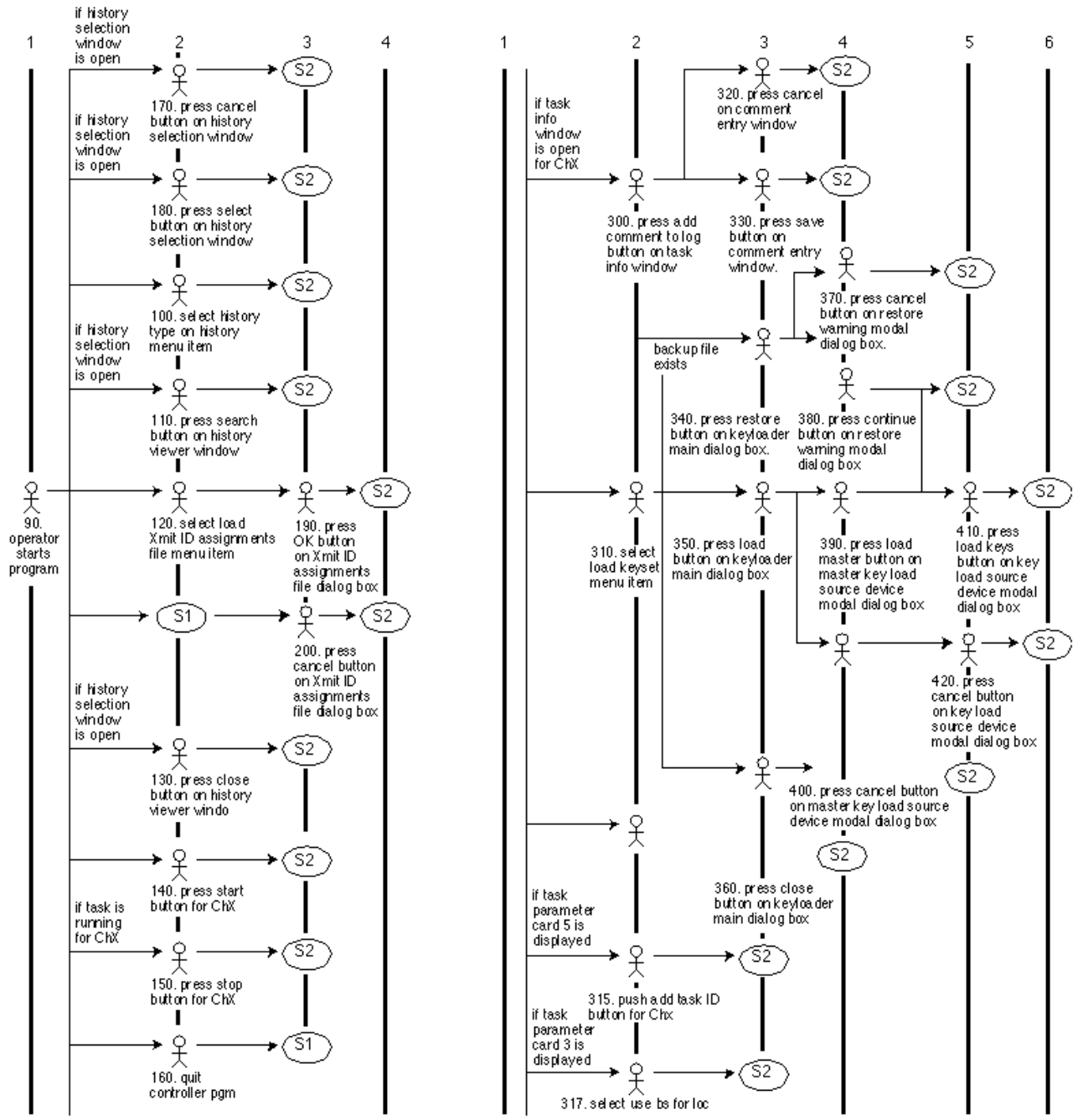

Figure 5

The actor-event diagram was most useful to systems engineers and the customer because it provided a simple, high-level picture of how the system worked before it was constructed or even designed. The diagram allowed developers and systems engineers to play the role of an actor by tracing through the diagram. It was also helpful to sketch out a GUI while building this diagram. When tracing through the diagram, engineers referenced the GUI that corresponded to each CMOS actor event. The customer also had no problem connecting the GUI to a sequence of actor events. More important, early role-playing allowed the development team to demonstrate the basic behavioral concepts of the system to our nontechnical customer in an understandable manner. The customer was able to suggest changes that made a significant difference in their satisfaction with the product prior to any development being completed.

Mayer Response Diagrams

The response diagrams were the most useful to software designers and implementers. Figures 6 and 7 are two samples of the 71 response diagrams that provide the details of how the software responds to the Mayer receiver actor events.

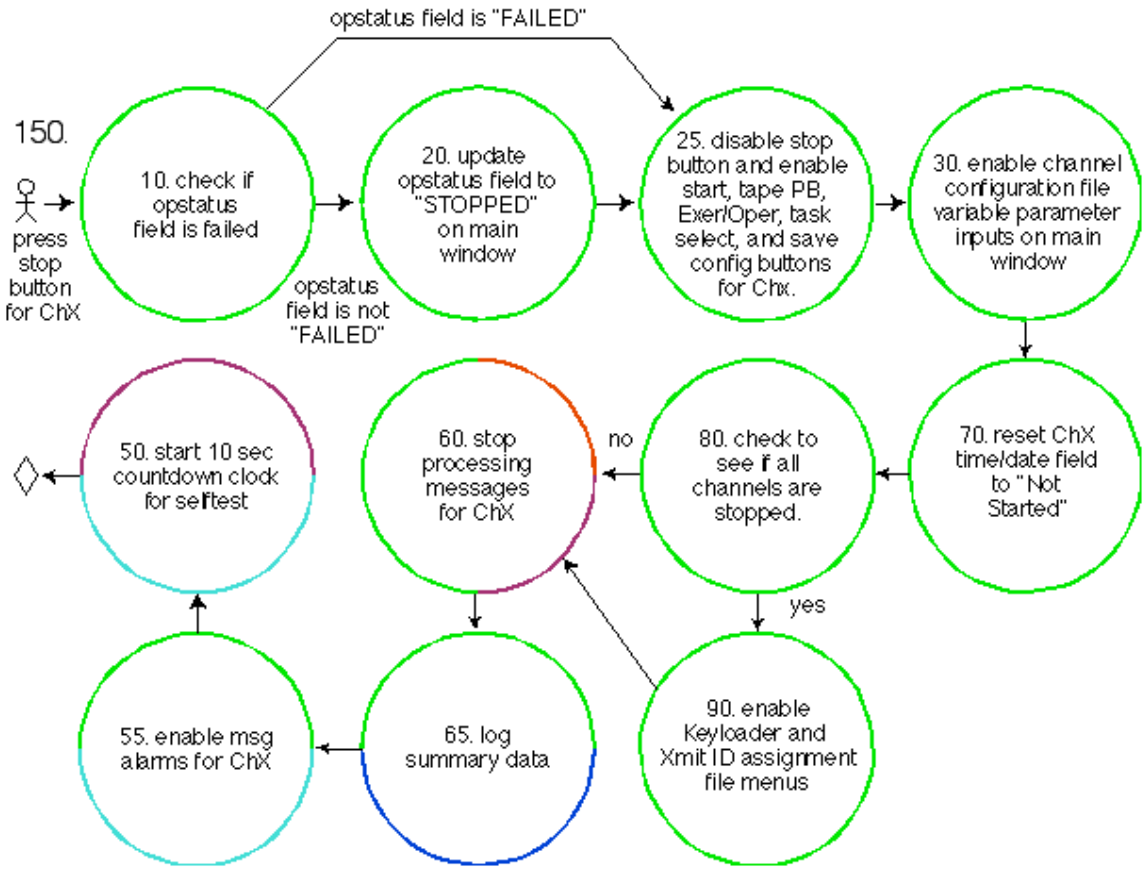

Figure 6

Each response diagram corresponds to an actor event defined on the actor diagram. Figure 6 represents the software responses to the actor event number 150, "Operator presses stop button for ChX." Figure 7 represents the software responses to actor event number 90, "Monitor starts program." The circles represent software responses that occur along possible scenario paths responding to the event. The colors (shades of gray in the black-and-white version) were added to represent 12 design domains that were created after the CMOS model was completed.

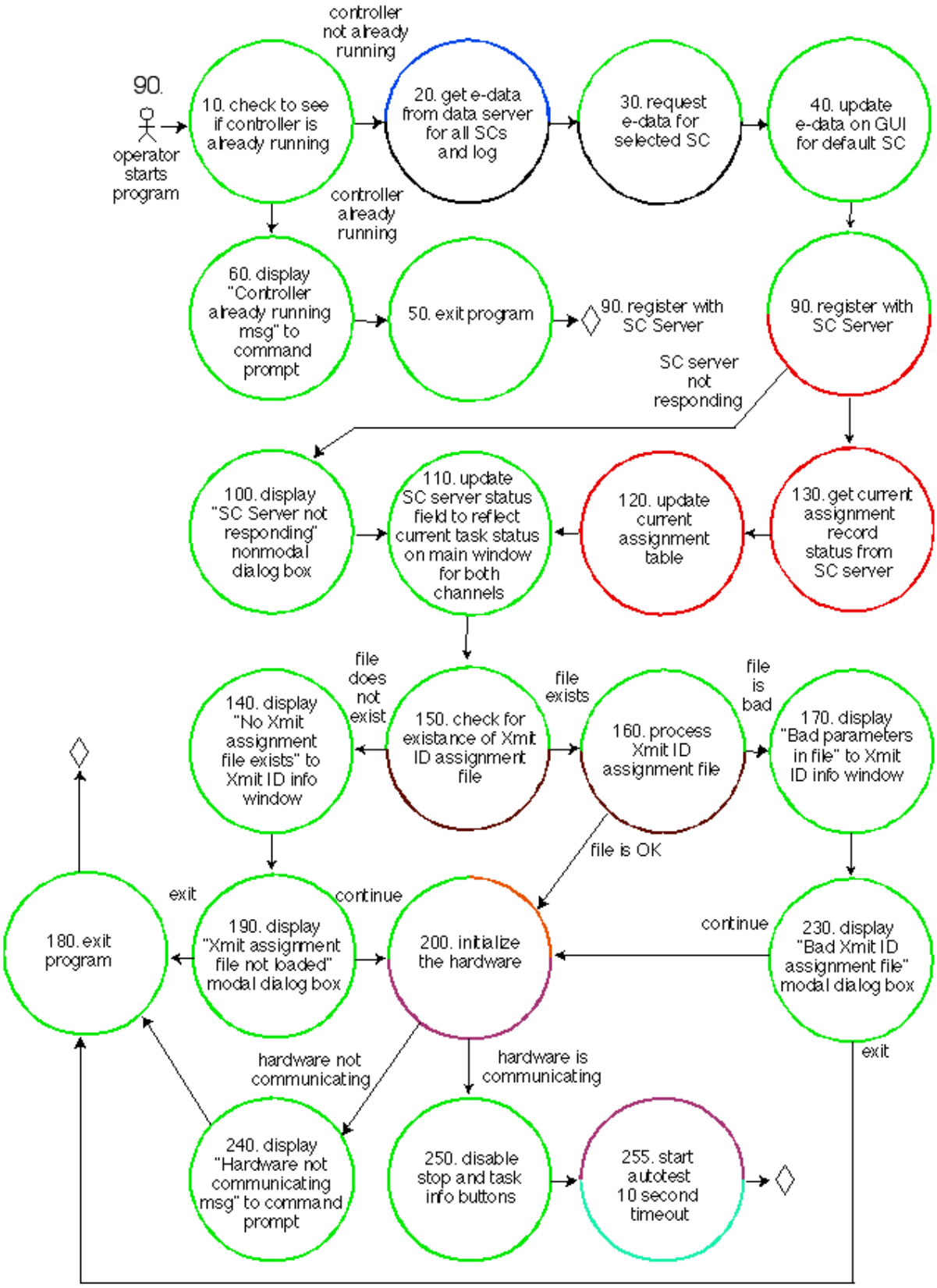
90.
operator starts program
10. check to see if controller is already running
controller not already running
20. get e-data from data server for all SCs and log
30. request e-data for selected SC
40. update e-data on GUI for default SC
controller already running
60. display "Controller already running msg" to command prompt
50. exit program
90. register with SC Server
90. register with SC Server
SC server not responding
100. display "SC Server not responding" nonmodal dialog box
110. update SC server status field to reflect current task status on main window for both channels
120. update current assignment table
130. get current assignment record status from SC server
file does not exist
140. display "No Xmit assignment file exists" to Xmit ID info window
150. check for existance of Xmit ID assignment file
file exists
160. process Xmit ID assignment file
file is bad
170. display "Bad parameters in file" to Xmit ID info window
file is OK
exit
180. exit program
continue
190. display "Xmit assignment file not loaded" modal dialog box
200. initialize the hardware
continue
230. display "Bad Xmit ID assignment file" modal dialog box
exit
hardware not communicating
hardware is communicating
240. display "Hardware not communicating msg" to command prompt
250. disable stop and task info buttons
255. start autotest 10 second timeout

Figure 7

Lessons Learned

From the Mayer receiver project, we learned a number of things about the use and usefulness of the CMOS methodology.

Numbering the Events and Responses. Even though actor events and responses have textual descriptions, it is helpful to attach number identifiers to them. This approach was not suggested in the original CMOS article. These numerical identifiers are helpful in that they:

- Make it easier to trace from actor events to system responses.
- Allow for tracking of work accomplished using actor events and responses.
- Let programmers easily identify actor events and responses in the code.
- Make it easier to reference actor events and responses when building test procedures or scenarios.

The numbering process uses intervals of 10 to allow insertion of future actor events or responses. The interval is arbitrary and can be any value; however, we recommend at least 10 to allow for future growth and modifications.

Splitting the Product into Domains. The first step in the software-design process was to compartmentalize the effort into application and service domains. These domains provide a set of related classes to perform a subset of the system functionality. By compartmentalizing the system in this manner, individual developers were able to take complete responsibility for a portion of the system. Additionally, it allows for prototyping and testing portions of the system without needing the entire system.

To do this, we studied the CMOS diagrams and decided how we could best group the many behaviors into some manageable grouping of applications to perform the system task. We identified 12 domains and assigned individual team members to each domain.

The key element of this effort was a domain communication model that was developed primarily from the CMOS model. The model uses color coding to easily identify each domain. Figure 8 is an example of the Mayer domain communication model.

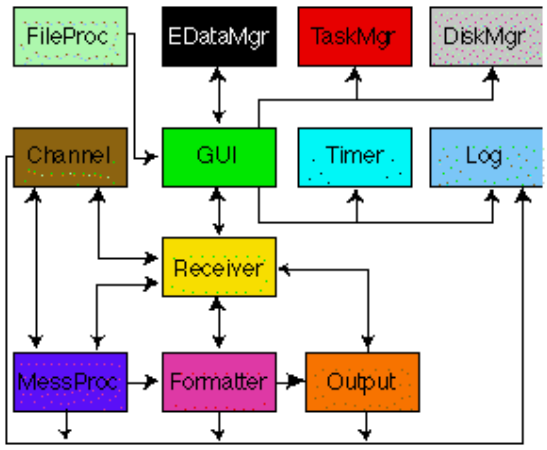

Figure 8

Overlaying Domains onto the Response Diagrams. Even though CMOS is a purely analytical method, we found that it also has the capability to reveal preliminary design information. We decided to indicate which domains participated in each CMOS response bubble so that developers would know for which ones they were responsible. A domain is a set of related classes. We used the domain's color code to overlay participating domains on response bubbles, making more visible which domains were participating in each bubble. In this process, we found that some bubbles had multiple domains participating. This is not too surprising because the bubbles represent behavior and the domains represent structure.

Next, the interfaces between the domains had to be defined before any serious domain-specific design could take place. We set out to build a domain communication model and quickly discovered that because we had identified each domain participating in a bubble, the bubbles with multiple domains (colors) revealed key locations in the design structure where messages would be passed (communication interfaces) between those domains. Essentially, the CMOS response diagram with the overlying domains showed exactly where the domains needed to communicate. This was a valuable bridge that helped us to relate the behavior in the CMOS model with the structure in the design's domain model.

Most design methodologies, including the Rational Unified Process and Shlaer-Mellor, suggest that a domain-level communication diagram be constructed to show the message communication between domains. CMOS helped this effort by providing domain communication paths early in the preliminary design phase. These communication paths were based on pure analysis of the system's behavior from the actor's perspective. The communication paths may not be complete, but should be correct. Completeness will come later because some of the domain communication paths will be established based on design or implementation considerations.

Tracking Progress with Response Diagrams. The response diagrams were used to track development progress -- specifically design, implementation, and testing efforts. Developers were assigned to the domains that overlaid the response diagrams. They were asked to estimate the hours it would cost to implement their portion of each bubble. Each bubble and its estimated hours were placed into a spreadsheet. (A total of all bubble's estimated hours was also a good source for costing the project.) As a developer worked on a bubble, he would enter a percent complete. Multiplying the percent complete by the estimated hours for each bubble determined hours spent and remaining for each bubble. Adding up all of the hours for each bubble determined hours spent and remaining for all of the bubbles combined. This allowed for an accurate percent-complete value that could be entered into our earned value system every month.

Creating Test Procedures. The CMOS actor-event and software-response diagrams allowed us to create test cases quickly and efficiently. Each actor event was listed numerically in a table. Figure 9 is a sample of the test case matrix for actor event number 90, "Operator starts Program." The first two columns track the test case to the CMOS actor event and its possible responses. Notice that under the "Software Scenarios" column each possible path (defined by response bubble numbers) is defined as a test case for actor event number 90. This allowed us to test the most probable scenario paths based on user behavior. Repeated paths were avoided to minimize the propagation of redundant cases. This was strictly an integration, black box test because it was based on behavior paths and not code paths. The entire set of test cases for every actor event was repeated three times; once for software integration testing, once for hardware integration testing, and finally for customer qualification testing. Using the CMOS model to produce these test cases was a great improvement

in test case production. Also, the test cases were produced just after the CMOS diagrams were completed and just before the design phase of the software began. By reviewing the test cases this early in the development cycle we were able to revalidate the requirements and reaffirm that our design was on target.

Test Case	Software Scenarios	Results	Procedure	Expected Results	Actual Results	Tester/Date
90. Operator starts program	10-60-50	90.00	Instructions on how to force this scenario path.	Describes what the tester should see when running this test case.	Tester enters what he actually saw.	Tester Name and date tested.
	10-20-30-40-90-100-110-150-140-190-180	90.01				
	10-20-30-40-90-100-110-150-160-170-230-180	90.02				
	10-20-30-40-90-100-110-150-160-200-250-255	90.03				
	10-20-30-40-90-100-110-150-110-200-150-180	90.04				
	10-20-30-40-90-100-110-150-160-200-240-180	90.05				

Placing Response Bubbles into the Code. The "Results" column in the test matrix for actor event 90 (Figure 9) is actually a directory. The directory contains a file for each of the 12 domains defined for the Mayer receiver software. Each file includes printouts of CMOS bubble numbers that were actually traversed when the software was run for that test case. This was accomplished by having each developer code an output statement that printed out the bubble number whenever the functionality of the bubble was executed by the code. This was extremely useful in determining if each test case actually followed the CMOS response path expected. It was also useful in regression testing because we could simply compare these files to later test case results to see if anything had changed. This technique was also valuable for locating errors in the code during the test. The test case would reveal which CMOS bubble did not work correctly. Developers would search (using a grep routine) for that bubble in the code. This process would lead developers to the exact location of the error.

Create a Drawing of the GUI Interface during Modeling. From the beginning of the modeling process it was helpful to draw GUI interface components as the behavior was being discovered. These drawings became invaluable when communicating behavior between developers and to the customer using the CMOS diagram. They were like visual aids to the model. It was not necessary to spend a great deal of time drawing these components. We used Microsoft Word to draw simple specifications of the components. It was a simple task to then build the real GUI components from the drawings.

Label GUI Components. Labeling GUI components with informative titles allows references to associated titles in the model. An example of this is circle 90.190 (Figure 7), which states "display xmit assignment file not loaded modal dialog box." The "xmit assignment file not loaded modal dialog box" is a title for a dialog box that was drawn in the GUI diagrams. The advantage of this approach is that the dialog box itself can change its graphical appearance without having to change the CMOS model.

Trace CMOS Scenarios at CDR. The Critical Design Review (CDR) for this project was a major success due in part to the CMOS method. We reversed the model into text scenarios and presented them along with the diagrams that produced them. The customer had no software technical

background and yet could easily read and understand the specified behavior. This let him make intelligent suggestions as well as further validating the behavior. An actual quote by the customer was "I have never seen a software CDR like this where I paid attention to every cell and I learned from it."

Usage with UML. We initially used Jacobson's Use Cases to help discover and record behavior for the Mayer receiver software. Use cases provided the same level of information as the actor-event diagrams. Use cases and the actor-event diagrams were useful in providing clarity for user behavior. The use cases and supporting interaction diagrams, however, did not provide enough information about how the software should respond to actor events. The CMOS system-response diagrams provided this information. After actor events were defined, the system-response diagrams were used to help bridge the software behavior to the software design (see "Overlaying the Domain Model onto the Response Diagrams"). Using use case diagrams by themselves would have been much more difficult. Following the construction of the domain model, the development team used UML object models and state diagrams to complete the design of the software.

It is interesting that the CMOS actor-event and response diagrams had some correlation with the UML state diagrams. For instance, we could map response bubbles that communicated between domains with the states that sent messages between those domains. All of the events in the actor-event diagrams mapped directly to events occurring in the state models. It was a simple matter to validate the state models by tracing events and messages back to the original behavior defined in the CMOS diagrams.

Suggested Improvements. Even though CMOS claims that it does not need an elaborate CASE tool, it would have been helpful to have some kind of mechanism to link actor events with responses. This capability became more important as the specification grew in size. A tool such as Visio, which provides this linking mechanism, would be better than Word for a larger specification. It also became cumbersome to try and draw the domain colors on the bubbles, especially when there were multiple domains on a bubble. These problems were minute compared to the benefits of using the method; however, it would be nice to see a CMOS-specific tool developed in the near future.

Conclusion

The specification produced by CMOS was invaluable throughout the software-development process. There were many occasions when we would reference the model to look up important behavioral and analysis data. Finding this data was easy using the model. Without it, we would have spent many extra hours writing and looking up textual representations for the same data. This alone accounted for a three-fold improvement in cycle time.

The model helped to greatly improve test-case development productivity. The test cases were developed early in the development cycle and were used to help validate the requirements before the design phase began. The model was also used as a validation tool when the customer added or changed requirements.

Interfaces between software domains were easily identifiable after overlaying the domain structure onto the CMOS response diagrams, saving a great deal of time and providing an accurate communication model.

Progress tracking of the development effort was made painless by both management and developers because the subjectivity was taken away. We tracked project tasks that, because of CMOS, had been

validated by management and customers. Developers only needed to know which bubbles they were working on and at the end of each week, filled in the percent complete on a simple spreadsheet.

Our customer was delighted with the specification because it was easy to read and communicated the expected software behavior in a clear and precise manner. Before a new or changed requirement was approved, it was first added to the CMOS model to make sure that it would work correctly with the existing system. This allowed developers to quickly determine if a new requirement was feasible.

The CMOS method is easy to learn and does not require a host of CASE tools to implement. Most any drawing tool can handle the CMOS modeling components. A mechanism to help link actor events with responses would have been helpful. Several future projects for the Mayer receiver are currently planning to use the CMOS method. For instance, the SSSD tools working group is using CMOS to help define how configuration and requirements management tools interface with various users.

We believe that this method can be used by anyone, and the time to incorporate the method is minimal, while the benefits are numerous.

References

Coats, Mark and Terry Mellon. "Constructing Operational Specifications," *DDJ*, June 1995. http://www.ddj.com/articles/1995/ 9506//documents/ddj9506a/.

Jacobson, Ivor et al. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, 1992.

Kruchten, P. *The Rational Unified Process: An Introduction*. Addison-Wesley, 1998.

Rumbaugh, James, et al. *Object-Oriented Modeling and Design*, Prentice Hall, 1991.

Shlaer, S. and S. Mellor. *Object Lifecycles: Modeling the World in States*, Prentice Hall, 1992.

CMOS Templates on COMPASS. http://compass.mot.com/ cgi?func=ll&objId=6468-2282&objAction=browse&sort=name.

DDJ
