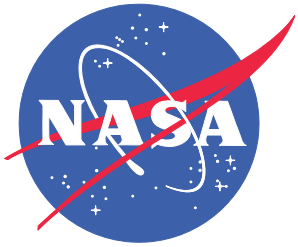
NASA/TM–2013-218031
NESC-RP-10-00609



Assess/Mitigate Risk through the Use of Computer-Aided Software Engineering (CASE) Tools

Michael L. Aguilar/NESC
Langley Research Center, Hampton, Virginia

August 2013

NASA STI Program . . . in Profile

Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA scientific and technical information (STI) program plays a key part in helping NASA maintain this important role.

The NASA STI program operates under the auspices of the Agency Chief Information Officer. It collects, organizes, provides for archiving, and disseminates NASA's STI. The NASA STI program provides access to the NASA Aeronautics and Space Database and its public interface, the NASA Technical Report Server, thus providing one of the largest collections of aeronautical and space science STI in the world. Results are published in both non-NASA channels and by NASA in the NASA STI Report Series, which includes the following report types:

- TECHNICAL PUBLICATION. Reports of completed research or a major significant phase of research that present the results of NASA Programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA counterpart of peer-reviewed formal professional papers, but having less stringent limitations on manuscript length and extent of graphic presentations.

- TECHNICAL MEMORANDUM. Scientific and technical findings that are preliminary or of specialized interest, e.g., quick release reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.

- CONTRACTOR REPORT. Scientific and technical findings by NASA-sponsored contractors and grantees.

- CONFERENCE PUBLICATION. Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or co-sponsored by NASA.

- SPECIAL PUBLICATION. Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest.

- TECHNICAL TRANSLATION. English-language translations of foreign scientific and technical material pertinent to NASA's mission.

Specialized services also include organizing and publishing research results, distributing specialized research announcements and feeds, providing information desk and personal search support, and enabling data exchange services.

For more information about the NASA STI program, see the following:

- Access the NASA STI program home page at *http://www.sti.nasa.gov*

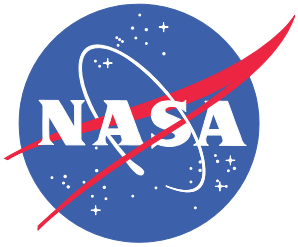
- E-mail your question to help@sti.nasa.gov

- Fax your question to the NASA STI Information Desk at 443-757-5803

- Phone the NASA STI Information Desk at 443-757-5802

- Write to:
  STI Information Desk
  NASA Center for AeroSpace Information
  7115 Standard Drive
  Hanover, MD 21076-1320

NASA/TM–2013-218031
NESC-RP-10-00609



Assess/Mitigate Risk through the Use of Computer-Aided Software Engineering (CASE) Tools

Michael L. Aguilar/NESC
Langley Research Center, Hampton, Virginia

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

August 2013

Acknowledgments

Significant Contributors to this report:

Mr. Julian C. Breidenthal; Systems Engineer; Jet Propulsion Laboratory

Mr. Ronald Morillo; Software Systems Engineer; Jet Propulsion Laboratory

Dr. Masoud Mansouri-Samani; IBM®, Rational®, and Rhapsody® System Architect; Ames Research Center

Mr. Kenneth M. Starr; CxP Systems Engineering; Jet Propulsion Laboratory

Available from:

NASA Center for AeroSpace Information
7115 Standard Drive
Hanover, MD 21076-1320
443-757-5802

Assess/Mitigate Risk through the Use of Computer-aided Software Engineering (CASE) Tools

July 11, 2013

Report Approval and Revision History

NOTE: This document was approved at the July 11, 2013, NRB. This document was submitted to the NESC Director on July 26, 2013, for configuration control.

Approved:	*Original Signature on File*	7/26/13
	NESC Director	Date

Version	Description of Revision	Office of Primary Responsibility	Effective Date
1.0	Initial Release	Mr. Michael L. Aguilar, NASA Technical Fellow for Software, GSFC	7/11/13

Table of Contents

Technical Assessment Report

List of Figures

Technical Assessment Report

1.0 Notification and Authorization

Mr. William Othon, Software Architect in the Constellation Program (CxP) Software Engineering and Integration Group, requested that the NASA Engineering and Safety Center (NESC) perform an independent technical assessment of NASA Program risk mitigation through the use of computer-aided software engineering (CASE) tools. The original plan contained two phases:

- Phase 1: Conduct a feasibility study to identify candidate solution paths to mitigate CxP risks.
- Phase 2: Demonstrate the processes and tools needed to provide system/software model analyses within and across tool sets.

The assessment plan for Phase 1 was approved by the NESC Review Board (NRB) on January 28, 2010. The plan for Phase 2 was approved by the NRB on April 29, 2010, and was subsequently modified and approved on May 26, 2011. Mr. Michael L. Aguilar, NASA Technical Fellow for Software at Goddard Space Flight Center (GSFC), was selected to lead this assessment.

The primary stakeholders for this assessment are Mr. William Othon for Phase 1 and Mr. Frank Bauer, Chief Engineer for the Exploration Systems Mission Directorate (ESMD), for Phase 2.

2.0 Signature Page

Submitted by:

Team Signature Page on File 8-7-13

Mr. Michael L. Aguilar Date

Significant Contributors:

Mr. Julian C. Breidenthal Date

Mr. Ronald Morillo Date

Dr. Masoud Mansouri-Samani Date

Mr. Kenneth M. Starr Date

Signatories declare the findings, observations, and NESC recommendations compiled in the report are factually based from data extracted from program/project documents, contractor reports, and open literature, and/or generated from independently conducted tests, analyses, and inspections.

3.0 Team List

Name	Discipline	Organization
Core Team		
Michael Aguilar	NESC Lead	GSFC
Jay Breidenthal	Systems Engineer	JPL
Tim Crumbley	NASA Technical Standards	MSFC
Lorraine Fesq	Systems Engineer	JPL
Stephanie Hamrick	MTSO Program Analyst	LaRC
Maddalena Jackson	MagicDraw®	JPL
Masoud Mansouri-Samani	IBM® Rational® Rhapsody ®, System Architect	ARC
Ron Morillo	Software Systems Engineer	JPL
Terry Morris	Electronics Engineer	LaRC
Bill Othon	Aerospace Engineer	JSC
Nicholas Rouquette	Object Management Group (OMG®)/UML	JPL
Marc Sarrel	Ground System Engineer	JPL
Kenneth Starr	CxP Systems Engineering	JPL
Consultants		
Jeff Jenkins	Integrated Model-Centric Engineering (IMCE) Project	JPL
Ed Siedewitz	Vendor	Model Driven Solutions
Administrative Support		
Linda Burgess	Planning and Control Analyst	LaRC/AMA
Jonay Campbell	Technical Writer	LaRC/NG
Diane Sarrazin	Project Coordinator	LaRC/AMA

4.0 Executive Summary

The NASA Engineering and Safety Center (NESC) was requested to perform an independent assessment of the Constellation Program (CxP) Risk 4421 mitigation through the use of computer-aided software engineering (CASE) tools. The stakeholder was Mr. William Othon, Software Architect for the CxP Software Engineering and Integration Group.

With the CxP cancellation, the assessment goals were modified to capture CASE tool usage lessons learned and best practices for future Programs and projects. The primary stakeholder for this modified (Phase 2) effort was Mr. Frank Bauer, Chief Engineer for the Exploration Systems Mission Directorate (ESMD).

4.1 Phase 1

The CxP was tracking Risk 4421:

> "Given the inherently complex software system being developed and the large number of stakeholders driving requirements and capabilities, there is a possibility that overly complex interfaces will result which impact test, integration, and operational activities."

The desired result from the Phase 1 effort was a recommended approach for policy, requirements, and/or guidance that should be applied at CxP Level 2 (L2) to ensure robust development of the CxP interfaces across the L2 to Level 3 (L3) boundary. The following questions were expected to be answered:

- Can the CASE tools in use by the CxP support the mitigation of this risk? (Assumptions and expectation in the use of the tools are untried.)
- Does a risk exist when the mitigation depends on specific CASE tools? (Vendor dependency is an issue, especially for long-term support.)

4.1.1 Could the risk CxP was tracking be mitigated given the tools, skills, process, and standards that were in place?

The assessment team determined that the current state-of-the-art CASE tools are capable of system modeling and analysis, but the available tools would not solve the complexity risk of the CxP L2 and L3 interfaces. The tools were expected to exchange models across the L2 and L3 interfaces, but this capability has not been successfully demonstrated. The verification of an exported-imported model is difficult, and the behavior of exported-imported models is not standardized within the Unified Modeling Language (UML) specification [ref. 1].

Distributed software modeling was successfully demonstrated in the James Webb Space Telescope (JWST) Integrated Science Instrument Module flight software development. The use of identical CASE development tools was a major contributor to the success by reducing the model export and import issues.

4.1.2 Can the CASE tools in use by CxP support the mitigation of this risk?

The assessment team explored assumptions and expectations in the use of CASE tools. Several models were developed from the CxP documentation. The assessment team demonstrated effective analysis and peer review of both static and executable models. The model-building process exposed issues with CxP document maturity, consistency, and completeness. The models were successfully developed from documents at different maturity levels. Interfaces were executed and analyzed nominally and with a single fault. The tool capability of partitioning and the hierarchical breakdown of state machines allowed complex concurrent processes to be modeled as separate, simpler models. This allowed complex systems to be modeled as a system of systems (SoS), with each element modeled and verified by appropriate subject matter experts and then peer reviewed.

4.1.3 Does a risk exist when the mitigation depends on specific CASE tools?

Software vendor dependency has been an issue, especially for long-term support. This assessment identified issues in tool-chain selection, interoperability, and use.

Generically, dependency on any specific development tool and/or vendor is an issue. Computer-aided design (CAD), computer-aided manufacturing, aeronautical databases, and software libraries have similar dependency issues. Exporting and importing databases or repositories requires considerable effort and verification.

This assessment demonstrated the building of executable models from documentation and from the static models of another CASE tool. This demonstration was performed manually, rather than through an export-to-import mechanism. These models were limited to a specific scope and focused on a specific issue or question. The assessment team determined that the selection of and dependence on a single tool chain, possibly from a single vendor, is the best choice when expecting to analyze the integration of components.

4.2 Phase 2

The assessment team planned to generate a final report with findings, observations, and NESC recommendations, as well as a model integration standard. This interim standard was to define the format and content of software designs and models so that the need for rework of these products by collaborating organizations would be minimized. Adherence to this standard would result in enhanced integration between organizations exchanging CASE products.

However, this interim standard was not produced. The assessment team determined that modification of the NASA Procedural Requirement (NPR) 7123.1A “NASA Systems Engineering Processes and Requirements” [ref. 2] and NPR 7150.2A “NASA Software Engineering Requirements” [ref. 3] to address interface issues and interface modeling issues would better benefit the system and software communities.

The rationale for not producing the interim standard was based on the following:

- The current NASA-STD-0007 "NASA Computer-aided Design Interoperability" [ref. 4] was experiencing difficulty in acceptance and adoption. It was not evident that it was effective.
- Numerous successful methods were evident in the current NASA Programs and projects, including manual, model, and automated methods.
- A review of the lessons learned identified a lack of defined processes and procedures to answer the question, "How are the system requirements at the interfaces being validated and verified?"
- NASA NPRs that define processes and procedures already exist and could be improved directly.

The NASA Program and project life cycle, described in reference 2, provides an overall framework for system development but is relatively unspecific as to what should be accomplished in software interface management. Each Program or project must make choices as to where, how, and when software interfaces are addressed, which can impact the ability to meet identified goals.

NPR 7123.1A emphasizes physical interfaces. If a software interface control document (ICD) exists, then it must be inferred from this NPR to be part of the interface type identified as "any other interface." The assessment team recommends that the NASA Office of the Chief Engineer (OCE) update NPR 7123.1A to specify the formal definition of ICD to include required software interface requirements.

NPR 7150.2A gives requirements on the representation of software interfaces in more detail, although it does not specify when these requirements must be met. The assessment team recommends that the OCE update the NPR 7150.2A software interfaces requirements to specify exactly when the requirements must be met and what constitutes the deliverable.

Attachment B in Appendix B of this report, "Checklist of Software Interface Management Activity by Life Cycle Phase," identifies the processes required to manage software interfaces by phase. Attachment A in Appendix B of this report, "Options for Software Interface Content," identifies the contents for interface documentation for different project types.

5.0 Assessment Plan

Mr. William Othon requested that the NESC perform an independent assessment of risk mitigation through the use of CASE tools. The objective of this effort was to determine the capabilities of several CASE tools used by the CxP in mitigating the inherent risks in this complex system. The review was to evaluate the techniques and tools required to create integrated cross-project software design models to provide L2 interface analysis and evaluation. At the time this assessment was initiated, the CxP had no direction on which model analysis was feasible across the CASE tools in use.

The basis for this assessment was CxP Risk 4421:

> "Given the inherently complex software system being developed and the large number of stakeholders driving requirements and capabilities, there is a possibility that overly complex interfaces will result which impact test, integration, and operational activities."

The original plan contained two phases:

- Phase 1: Conduct a feasibility study to identify candidate solution paths to mitigate CxP risks.
- Phase 2: Demonstrate processes and tools needed to provide system/software model analysis within and across tool sets.

The scope was modified after the CxP was cancelled and the assessment team felt that the lessons learned from Phase 1 should not be lost. Therefore, Phase 2 was initiated to capture the CxP lessons for future Programs and projects, and Mr. Frank Bauer, ESMD Chief Engineer, was added as the primary stakeholder.

The Phase 2 activities included:

- Demonstrate the processes and tools needed to provide system/software model analysis within and across tool sets.
- Capture the CxP lessons learned in the NESC final report.
- Develop interim standard document, "NASA Computer-aided Software Engineering Integration."
    - Model this standard using the current NASA-STD-0007, "NASA Computer-aided Design Interoperability."
    - Establish the format and content of designs and models so that rework of these products by collaborating organizations is minimized. Adherence to this standard will result in enhanced integration between organizations exchanging CASE products.

The Phase 2 assessment plan was modified as a result of:

- The loss of the assessment's primary stakeholder when the CxP was cancelled.
- NESC funding availability.
- Higher priority efforts involving a number of key assessment team members.
- Increased external contracting of software and subsystems, which challenges oversight of design and development.

The NESC formed an assessment team with relevant expertise to review the CxP CASE tools. This assessment team consisted of Agency subject matter experts with systems perspective to develop a recommended approach for policy, requirements, and/or guidance that could be applied at CxP L2 to ensure robust development of L2-to-L3 boundary interfaces.

6.0 Phase 1

6.1 Phase 1: Data Analysis

Phase 1 involved the following activities:

1. Survey of the tools and processes used by the CxP Orion and Ares teams, the JWST Program, and Lockheed Martin.
2. Examination of the relevant experiences with the use of these tools at lower level projects (i.e., CxP L3: Orion and Ares).
3. Determination of whether the state-of-the-art CASE tools could simplify definition and design of the CxP L2 interfaces.
4. Identification of issues and gaps in the ICD and design documentation.
5. Identification of issues and successes associated with the use of these tools.
6. Development of a list of major findings, observations, and NESC recommendations.

After surveying the maturity of the document set and the CASE tool modeling efforts, the assessment team focused on demonstration of the strengths and weaknesses of the modeling tools. Assumptions about how the CxP expected these tools and models to mitigate the complex interface issues were assessed against tool capabilities. Modeling and analysis within one CxP development team was shown to be highly successful where all interfaces were within the scope of the single model. Modeling and analysis across team boundaries was successful in discovering interface errors between the teams, as long as the teams supported model sharing in the development plans and processes. Modeling across the CxP contractor team boundaries was not possible to demonstrate; no plan or process existed to support model exchange, export, and

import between disparate CASE tools, and the veracity of model importing and exporting between tools could not be determined.

During Phase 1, demonstrations of CASE tool capabilities were conducted. Translations from the CxP documentation to executable models were performed to demonstrate the analysis capabilities of the executable model. For example, in Figure 6.1-1, the communication protocol between the Orion and Ares projects was modeled and analyzed.

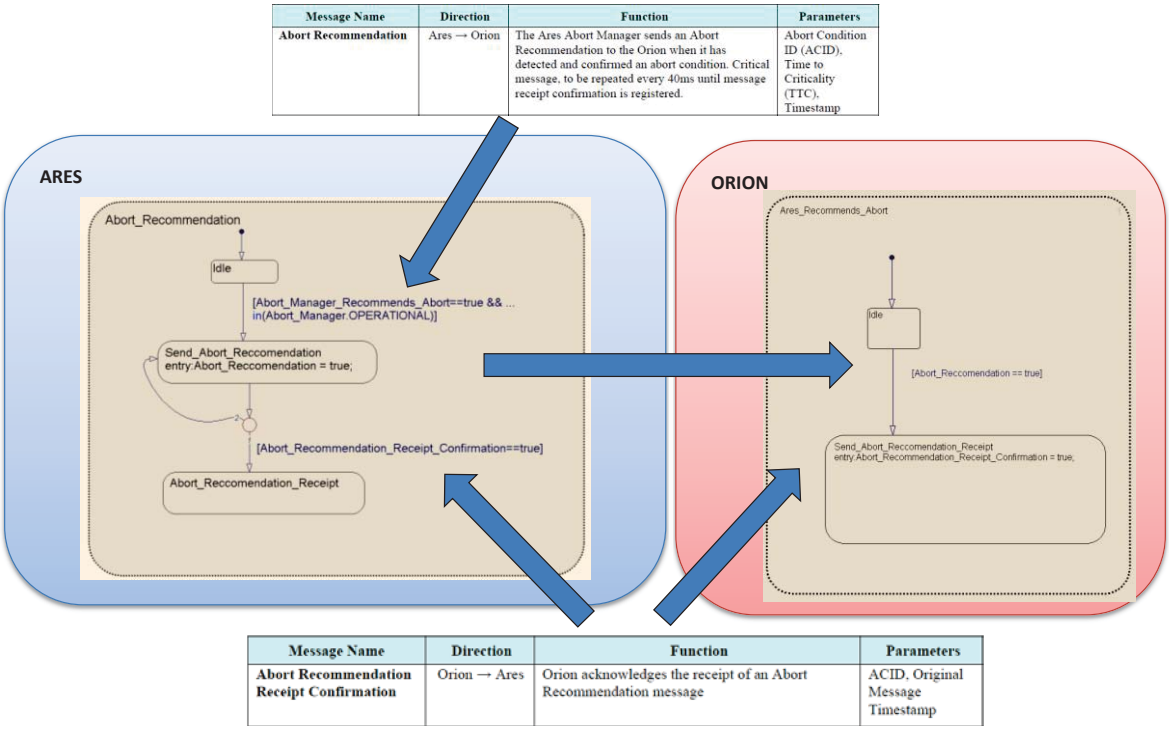


Figure 6.1-1. Example of Document to Executable Model Development

In another demonstration, static models developed in one CASE tool were translated into executable models using another CASE tool. For example, in Figure 6.1-2 the launch countdown timeline was modeled as activities, external states, and vehicle states. The executable model was then analyzed.

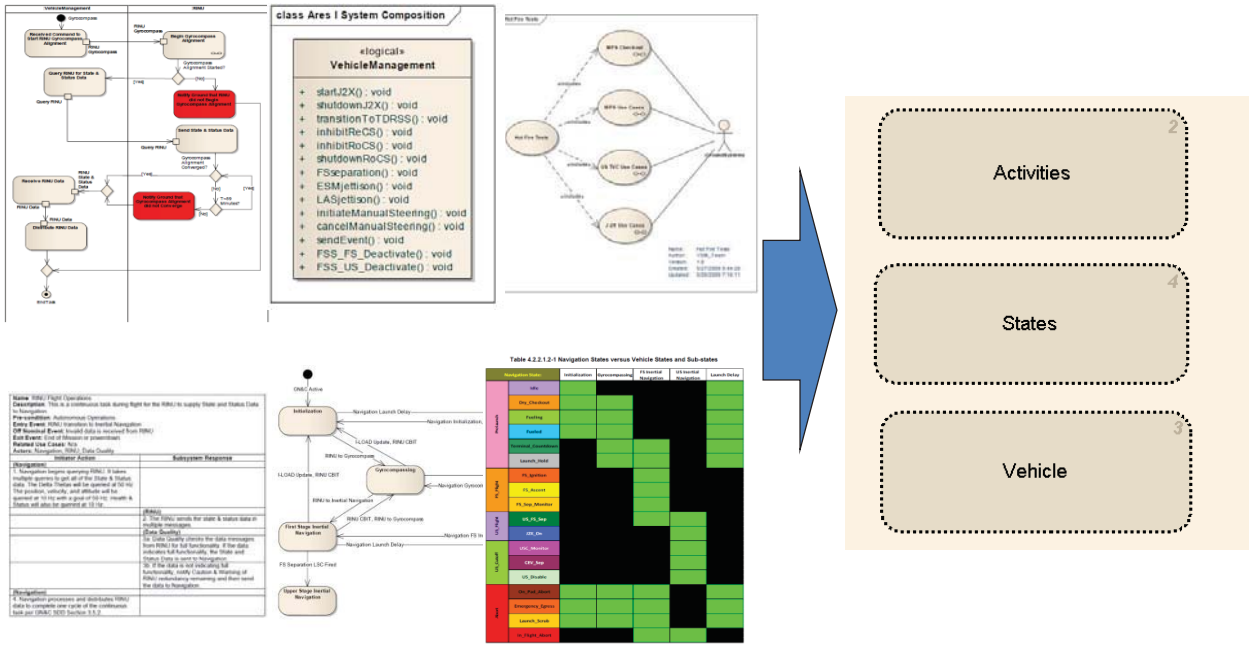


Figure 6.1-2. Example of Model to Executable Model Development

6.2 Phase 1: Reporting

A summary of the Phase 1 accomplishments is contained in Appendix A. The findings, observations, and NESC recommendations resulting from Phase 1 are given in Section 8.1.

7.0 Phase 2

7.1 Phase 2: Data Analysis

Phase 2 included the following activities:

1. Identification of pivotal reports in the literature.
2. Identification of necessary conditions for successful software integration.
3. Identification of software integration issues associated with various aspects of SoS development.
4. Review of the CxP lessons learned.
5. Review of state of the practice in model-centric engineering.
6. Selection of lessons learned and best practices that address the identified issues.

7. Identification of NASA requirements and standards that need to address these best practices and issues.

Phase 2 dealt mainly with a survey of the CxP lessons learned. Current practices were reviewed (see the 65 references listed in Appendix B). The selection process identified software interfaces as the most viable point to manage software complexity.

The essence of software development is "divide and conquer." A Program is partitioned into smaller projects, which in turn breaks problems into smaller parts. Each of these smaller problems is broken down until the problem can be solved with source code.

The assessment team developed a checklist of software interface activities to perform over the full systems engineering life cycle, which is contained in Attachment B of Appendix B. This checklist is summarized as:

- Problem definition
- Design
- Integration
- Verification and validation (V&V)
- Concept studies
- Concept and technology development
- Preliminary and final design
- System assembly, integration, test, and launch
- Operations, sustainment, and closeout

The assessment team developed a set of example software interface document contents to support life cycle activities, which is given in Attachment A of Appendix B.

The assessment team considered the implications of the developing state of the art in model-centric engineering. The complexity of software development surpasses the capabilities of capturing the interfaces in a set of documents, spreadsheets, and illustrations. The current CASE tools can capture interface details and behavior across interfaces with capabilities beyond printed documentation.

7.2 Phase 2: Reporting

A summary of work accomplished in Phase 2 is contained in Appendix B. The findings, observations, and NESC recommendations resulting from Phase 2 are given in Section 8.2.

8.0 Findings, Observations, and NESC Recommendations

The findings, observations, and NESC recommendations are presented according to assessment phase.

8.1 Phase 1

8.1.1 Phase 1 Findings

The following findings were identified during Phase 1:

F-1. No single tool-chain was selected by the CxP Ares project. Each subproject used its own tools. (See Section A-4.4 in Appendix A.)

F-2. No single software modeling standard was selected by the CxP Ares project. Each subproject defined its own standards. (See Section A-4.4 in Appendix A.)

F-3. CxP teams used tools that could not enforce consistency or establish traceability among different CASE models. (See Section A-4.6.3 in Appendix A.)

F-4. Teams used tools that could not establish traceability among software artifacts to the level of detail necessary to track changes across interfaces. (See Section A-3.2.1 in Appendix A.)

F-5. CxP teams used CASE tools that were not fully integrated to support the software life cycle. (See Section A-4.3 in Appendix A.)

F-6. Software design models (e.g., ICDs, software design documents (SDDs)) were often not available in an analyzable form. Diagrams were created using tools such as Microsoft® PowerPoint®, Microsoft® Visio, and Dia. (See Section A-4.4 in Appendix A.)

F-7. Some subprojects failed to provide the developers with timely access or sufficient training. (See Section A-4.4.1.3 in Appendix A.)

F-8. Not all of the tools used for modeling could support model simulation or animation (e.g., Microsoft® PowerPoint®, Microsoft® Visio, and Enterprise Architect (EA)). (See Section A-2.1 in Appendix A.)

8.1.2 Phase 1 Observations

The following observations were identified during Phase 1:

O-1. Problems existed with communication, model sharing, and reuse among different teams due to lack of model plan and processes. (See Sections A-4.4 and A-4.7.5 in Appendix A.)

O-2. Inconsistencies existed between model artifacts from different tools (e.g., an SDD or ICD could contain different/inconsistent diagrams). (See Sections A-3.2 and A-4.2 in Appendix A.)

O-3. It was difficult to establish traceability between different artifacts (e.g., requirements and design elements). (See Sections A-4.3 and A-4.7.2.1 in Appendix A.)

O-4. Advanced V&V tools could not be applied to models early in the life cycle, where requirement or design bugs could go unnoticed until the implementation and testing stages. (See Section A-4.3.2 in Appendix A.)

O-5. It was difficult to assess whether there were gaps in the overall design or the requirements. (See Section A-5.1.4 in Appendix A.)

O-6. Keeping consistency and maintaining traceability was a manual and time-consuming process. (See Section A-4.4 in Appendix A.)

O-7. Team members continued using different diagrams and models generated by numerous tools, often with different semantics, preventing an integrated design of components. (See Section A-4 in Appendix A.)

O-8. Tool capabilities were not fully exploited (e.g., requirements management and traceability could have been managed by the CASE tool, or model validation rules could have been defined). (See Section A-4.5 in Appendix A.)

O-9. The inability to simulate or animate models was a source of uncertainty and ambiguity, particularly during reviews where behavioral models were inspected. (See Section A-4.4 in Appendix A.)

8.1.3 Phase 1 NESC Recommendations

The following NESC recommendations were directed to the CxP and were presented to the CxP modeling team. As a result of the cancellation of the CxP during Phase 1, the following NESC recommendations were redirected to Mr. Frank Bauer, ESMD Chief Engineer:

R-1. Use CASE tools for model-driven software development as part of an integrated tool-chain that supports the entire software life cycle. ***(F-1, F-3, F-4, F-5, O-1, O-2, O-3, O-5, O-6, O-7)***

R-2. Establish processes, metrics, and a set of standards, guidelines, and best practices for effective model-driven software development. ***(F-2, O-1, O-2, O-7)***

R-3. Use a tool-chain that supports consistency and traceability between different artifacts. ***(F-2, F-3, F-4, O-2, O-3, O-5, O-6)***

R-4. Use tools that allow the user to create and maintain models in an analyzable form. ***(F-6, F-8, O-2, O-4, O-5, O-6, O-9)***

R-5. Reduce uncertainty and ambiguity by using executable models that provide animation and simulation. ***(F-8, O-2, O-4, O-5, O-9)***

R-6. Train personnel in the effective use of CASE tools. ***(F-7, O-1, O-7, O-8)***

8.2 Phase 2

8.2.1 Phase 2 Findings

The following findings were identified during Phase 2:

F-9. NPR 7123.1A emphasizes physical interfaces, with little emphasis on software interfaces. (See Section B-1.5.3 in Appendix B.)

F-10. NPR 7150.2A gives requirements on representation of software interfaces in more detail, although it does not specify when the requirements must be met. (See Section B-1.5.3 in Appendix B.)

F-11. Experience with the CxP revealed that Program-level integration suffered from the lack of timely and accurate design descriptions, especially in support of key milestone reviews. (See Section B-2.1.1.1 in Appendix B.)

F-12. The Soil Moisture Active Passive (SMAP) project source material contained inconsistencies that were discovered using models. (See Section B-2.1.1.4 in Appendix B.)

F-13. Having multiple teams working on a common system model encouraged agreement on terminology and logical decomposition. (See Section B-2.1.1.4 in Appendix B.)

F-14. The SMAP project used a system model to describe spacecraft system interfaces in various levels of refinement, ranging from a subsystem- to a low-level view focused on electrical system connections. (See Section B-2.1.2.3 in Appendix B.)

F-15. The European Southern Observatory (ESO) is producing ICDs based on a system model. (See Section B-2.1.2.3 in Appendix B.)

F-16. The European Space Agency (ESA) uses functional system interface simulators and end-to-end behavior for design and performance verification, as well as subsystem and payload V&V. (See Section B-2.1.2.3 in Appendix B.)

F-17. The Florida Institute of Technology is training students to develop a model-based systems engineering (MBSE) flight system model focusing on electrical systems interfaces, with follow-on projects planned, advancing into to full project deliverables leading to a CubeSat launch. (See Section B-2.1.2.3 in Appendix B.)

F-18. The Exploration Development Integration Office is using integrated functional analysis to describe interfaces, off-nominal situations, aborts, hazards, and safety and mission assurance issues, including interfaces for all mission configurations. (See Section B-2.1.2.3 in Appendix B.)

F-19. The Extravehicular Activity (EVA) project is modeling the interfaces on EVA space suits. (See Section B-2.1.2.3 in Appendix B.)

F-20. The Integrated Power Avionics and Software (iPAS) project is using MBSE to specify interfaces of components in their "Common Vehicle Architecture" catalog. (See Section B-2.1.2.3 in Appendix B.)

F-21. The CxP Orion System Architecture Model included vehicle interface and interconnect definitions. (See Section B-2.1.2.3 in Appendix B.)

F-22. The CxP encountered problems with a bottom-up design that caused ICDs to focus on low-level details, creating unnecessary duplication across ICDs and interface requirements documents. (See Section B-2.1.2.4 in Appendix B.)

F-23. The ESA has been pursuing the capability for concurrent engineering in a multidisciplinary environment, with the goals of integrating customer, engineering team, tools, project data, and mission and system models and enabling simultaneous participation of all mission domains, including cost engineering, risk analysis, programmatics, operations, CAD, and simulation. (See Section B-2.1.3.1 in Appendix B.)

F-24. The CxP attempted to develop a detailed (roll-up) integrated mission timeline in the preliminary design review timeframe from a number of system-specific timelines, which found various inconsistencies. (See Section B-2.1.3.1 in Appendix B.)

F-25. iPAS provided a standardized environment for hardware/software evaluation and test. (See Section B-2.1.3.3 in Appendix B.)

F-26. A current capability of one NASA model pulls together Space Launch System functions (modeled in EA) to the Exploration Systems Directorate capabilities (modeled in 3SL Cradle) to create an up-to-date integrated functional analysis report. (See Section B-2.1.4.2 in Appendix B.)

F-27. The CxP established numerous compartmentalized sites for access to various tools, data directories, and technical review repositories, each with local access rules. (See Section B-2.2.3 in Appendix B.)

F-28. The CxP encountered difficulty with nested interfaces at the top level resulting loss of information in the high-level models, which made it impossible to relate the lower level interfaces to higher level requirements. (See Section B-3.1.1.5 in Appendix B.)

F-29. The CxP's fractionated efforts at modeling and conversion into artifacts led to numerous engineering incompatibilities. (See Section B-3.1.5 in Appendix B.)

8.2.2 Phase 2 Observations

The following observations were identified during Phase 2:

O-10. The use of machine-usable models creates major possibilities not accessible to non-model-based engineering, specifically:

- Checking models for certain types of completeness, consistency, and accuracy.
- Exchanging information between models and modelers.
- Maintaining consistency between models by identifying and correcting inconsistencies and by efficiently propagating changes.
- Deriving system characteristics (e.g., performance, behavior, resource usage, and/or cost).

(See Section B-1.4 in Appendix B.)

O-11. The NASA project life cycle requires a preliminary awareness of software by the mission concept review and a substantial awareness—to the point of knowing what software needs to be acquired and managed for configuration—by the software requirements review. (See Section B-1.5.3 in Appendix B.)

O-12. Using models, software interfaces may be identified at higher levels in the system hierarchy than the level at which they are eventually implemented. (See Section B-1.6.6 in Appendix B.)

O-13. It is necessary to demonstrate the benefits of system modeling through:

- Simulation, validation, and model transformation
- Reuse of design elements
- Ability to generate multiple artifacts (e.g., documentation, code)
- Model transformations that allow use of capabilities of different tools
- Consistency and correctness across artifacts (e.g., documents, models, products)

(See Section B-2.1.1.4 in Appendix B.)

O-14. Modeling fault conditions and off-nominal scenarios is an area of needed research. These models need to include hardware, software, and human factors components. (See Section B-2.1.3.1 in Appendix B.)

O-15. No examples of practical work that include the integration of system models, control engineering models, and control software models were identified based on a state analysis paradigm. This modeling integrates flight mechanics, GN&C, hardware, software and other disciplines that deal with real-time control performance issues.

Little has been done trying to model the "as-built" performance before implementation. (See Section B-2.1.3.5 in Appendix B.)

O-16. De-scoping MBSE tool capabilities and environments can cause failures in the modeling and tool usage. (See Section B-2.1.5.4 in Appendix B.)

O-17. Using models, black-box interactions can be defined between a system and its environment or between system timelines. (See Section B-3.1.1.1 in Appendix B.)

O-18. The black-box models can be tested against parties in the environment by simulating system activity. (See Section B-3.1.3 in Appendix B.)

O-19. A CxP lesson learned was to avoid fractionated efforts at modeling and conversion into artifacts because of the engineering incompatibilities. (See Section B-3.1.5 in Appendix B.)

O-20. Model-based training should be tailored to need and usage. (See Section B-3.1.5 in Appendix B.)

O-21. Software system functions (e.g., shutdown, start-up, and fault management functions) should be derived, modeled, and analyzed. (See Section B-3.3.4 in Appendix B.)

O-22. The use of an industry-recognized standard for system interfaces reduces the complexity of the interface model. (See Section B-3.3.7 in Appendix B.)

O-23. Configuration management must handle coordination and synchronization between higher and lower level models and interfaces. (See Section B-4.3 in Appendix B.)

O-24. The ESO observed that consistency and correctness across artifacts is a significant driver on the cost of verification. (See Section B-4.4 in Appendix B.)

O-25. The completeness, correctness, and clarity of the interface requirements on the SoS level can be evaluated, along with emergent characteristics associated with external and internal software interfaces, using analyzable models. (See Section B-4.4 in Appendix B.)

8.2.3 Phase 2 NESC Recommendations

The following NESC recommendations are directed to the ESMD Chief Engineer:

R-7. Update NPR 7123.1A to specify the formal definition of ICDs to include the details required of software interfaces. ***(F-9, F-11, F-22, F-27, F-28)***

R-8. Update the software interfaces requirements in NPR 7150.2A to specify exactly what and when the interface requirements must be met***. (F-9, F-10, F-11, F-22, F-27, F-28, F-30, O-11, O-12)***

R-9. Update NPR 7150.2A software interfaces requirements to incorporate Attachments A and B from Appendix B of this report. ***(F-9, F-10, F-11, F-22, F-27, F-28, O-11, O-12)***

9.0 Alternate Viewpoint

There were no alternate viewpoints identified during the course of this assessment by the NESC team or the NRB quorum.

10.0 Other Deliverables

No unique hardware, software, or data packages, outside those contained in this report, were disseminated to other parties outside this assessment. All model analysis resulting in the possibility system errors in design were shared with CxP during Phase 1.

11.0 Lessons Learned

No applicable lessons learned were identified for entry into the NASA Lessons Learned Information System (LLIS) as a result of this assessment.

12.0 Recommendations for NASA Standards and Specifications

See the Phase 2 NESC recommendations in Section 8.2.3.

13.0 Definition of Terms

CASE Tool	Provides automated assistance for software development.
Corrective Actions	Changes to design processes, work instructions, workmanship practices, training, inspections, tests, procedures, specifications, drawings, tools, equipment, facilities, resources, or material that result in preventing, minimizing, or limiting the potential for recurrence of a problem.
Executable Model	Models with a behavioral specification precise enough to be effectively executed and analyzed
Finding	A relevant factual conclusion and/or issue that is within the assessment scope and that the team has rigorously based on data from their independent analyses, tests, inspections, and/or reviews of technical documentation.
Lessons Learned	Knowledge, understanding, or conclusive insight gained by experience that may benefit other current or future NASA Programs and projects. The experience may be positive, as in a successful test or mission, or negative, as in a mishap or failure.

Observation	A noteworthy fact, issue, and/or risk, which may not be directly within the assessment scope, but could generate a separate issue or concern if not addressed. Alternatively, an observation can be a positive acknowledgement of a Center/Program/Project/Organization's operational structure, tools, and/or support provided.
Problem	The subject of the independent technical assessment.
Proximate Cause	The event(s) that occurred, including any condition(s) that existed immediately before the undesired outcome, directly resulted in its occurrence and, if eliminated or modified, would have prevented the undesired outcome.
Recommendation	A proposed measurable stakeholder action directly supported by specific Finding(s) and/or Observation(s) that will correct or mitigate an identified issue or risk.
Root Cause	One of multiple factors (events, conditions, or organizational factors) that contributed to or created the proximate cause and subsequent undesired outcome and, if eliminated or modified, would have prevented the undesired outcome. Typically, multiple root causes contribute to an undesired outcome.
Supporting Narrative	A paragraph, or section, in an NESC final report that provides the detailed explanation of a succinctly worded finding or observation. For example, the logical deduction that led to a finding or observation; descriptions of assumptions, exceptions, clarifications, and boundary conditions. Avoid squeezing all of this information into a finding or observation
UML	Used to graphically specify, visualize, and document models of software systems.

14.0 Acronyms List

ARC	Ames Research Center
CAD	Computer-aided Design
CASE	Computer-Aided Software Engineering
CxP	Constellation Program
EA	Enterprise Architect
ESMD	Exploration Systems Mission Directorate
ESO	European Southern Observatory
EVA	Extravehicular Activity
GSFC	Goddard Space Flight Center
ICD	Interface Control Document

IMCE	Integrated Model-centric Engineering
iPAS	Integrated Power Avionics and Software
JPL	Jet Propulsion Laboratory
JSC	Johnson Space Center
JWST	James Webb Space Telescope
L2	Level 2
L3	Level 3
LaRC	Langley Research Center
MBSE	Model-Based Systems Engineering
MSFC	Marshall Space Flight Center
NASA	National Aeronautics and Space Administration
NESC	NASA Engineering and Safety Center
NPR	NASA Procedural Requirement
NRB	NESC Review Board
OMG	Object Management Group
SDD	Software Design Document
SMAP	Soil Moisture Active Passive
SoS	System of Systems
UML	Unified Modeling Language
V&V	Verification and Validation

15.0 References

1. "Documents Associated with the Unified Modeling Language (UML), v2.4.1," Object Management Group, August 2011. URL: http://www.omg.org/spec/UML/2.4.1/, accessed July 8, 2013.
2. "NASA Systems Engineering Processes and Requirements," NASA Procedural Requirement (NPR) 7123.1A, NASA Office of the Chief Engineer, March 26, 2007.
3. "NASA Software Engineering Requirements," NASA Procedural Requirement (NPR) 7150.2A, NASA Office of the Chief Engineer, November 19, 2009.
4. "NASA Computer-aided Design Interoperability," NASA-STD-0007, NASA Technical Standards Program, June 1, 2009.

16.0 Appendices

Appendix A. Phase 1 Report

Appendix B. Phase 2 Report

Appendix A. Phase 1 Report

Phase I: Assess the Mitigation of Risks Through the Use of Computer-Aided Software Engineering (CASE) Tools

TI-10-00609 (Phase 1)

Table of Contents

Technical Assessment Report

List of Figures

List of Tables

A-1.0 Executive Summary

The NASA Engineering and Safety Center (NESC) was requested to perform an independent assessment of the mitigation of risks through the use of Computer-Aided Software Engineering (CASE) tools.

The Constellation Program (CxP) was tracking CxP risk 4421:

> Given the inherently complex software system being developed and the large number of stakeholders driving requirements and capabilities, there is a possibility that overly complex interfaces will result which impact test, integration, and operational activities.

The requestor was looking for a recommended approach for policy, requirements, and/or guidance that could be applied at the CxP Level 2 (L2) to ensure robust development of the CxP interfaces across the L2 to Level 3 (L3) boundary. The following questions were to be answered:

- Can the CASE tools in use by CxP support the mitigation of this risk? (Assumptions and expectations in the use of the tools are untried.)
- Does a risk exist when the mitigation depends on specific CASE tools? (Vendor dependency is an issue, especially for long-term support.)

In completing this assessment, the following activities were conducted:

1. A survey of the tools and processes used by Orion, Ares, the James Webb Space Telescope (JWST), and Lockheed Martin.
2. Identification of issues and gaps in interface control documentation and design documentation.
3. Examination of the relevant experiences with the use of these tools at lower level projects (i.e., CxP L3: Orion and Ares).
4. Based on the survey results, a determination of whether the state-of-the-art CASE tools can help simplify definition and design of interfaces at CxP L2.
5. Identify the issues as well as success stories associated with the use of these tools.
6. Provide a list of major findings, issues, and recommendations.

A-1.1 CASE Tool Risk Mitigation Concerns and Issues

The assessment team identified the following concerns and issues.

No single tool chain was selected by the Ares project. No single software modeling standard was selected by the Ares project.

Teams often used tools that could not enforce consistency among different model representations. Teams often used tools that could not establish traceability among different artifacts or that were not fully integrated to support the entire software life cycle. Some subprojects selected certain CASE tools but failed to provide the developers with access or sufficient training in a timely manner.

Software design models included in the documents under review (e.g., interface control documents (ICDs), software design documents (SDDs)) presented models as pictures developed in Microsoft® PowerPoint®, Microsoft® Visio®, and Dia. These models were not available in an analyzable form. Not all of the tools used for modeling could support model simulation or animation (e.g., PowerPoint®, Visio®, Enterprise Architect).

Problems existed with communication, model sharing, and reuse among different teams. Inconsistencies existed between model artifacts from different tools. An SDD or ICD could contain different diagrams, which were inconsistent. It was difficult to establish traceability between different artifacts (e.g., requirements and design elements).

The teams could not apply advanced validation and verification (V&V) tools to models early in the life cycle. Requirements or design bugs could go unnoticed until the implementation and testing stages, making them more costly to fix. The selected tools made it difficult to assess whether gaps existed in the overall design or the requirements. Maintaining consistency and full traceability was not supported by the tools and became a manual and time-consuming process.

The full capabilities of given tools was not fully exploited; for example, requirements management and traceability could have been managed by the CASE tool, or model validation rules could have been defined. Lacking the capability to simulate or animate models was a source of uncertainty and ambiguity, particularly during reviews where behavioral models were inspected. It was difficult to visualize model behavior and assess the full impact of changes.

A-1.2 Could the risk CxP was tracking be mitigated given the tools, skills, processes, and standards that were in place?

The risk that complex software system interfaces being developed would impact test, integration, and operational activities could not be mitigated using the tools and processes in place during the CxP.

A-1.2.1 Could the CASE tools in use by CxP support the mitigation of this risk?

Assumptions and expectations in the use of the tools were tested during the course of this assessment. An integrated tool chain, along with support for the tool chain, could have mitigated the risk. The current tools have the capabilities required to mitigate the risk inherent in complex systems.

A-1.2.2 Does a risk exist when the mitigation depends on specific CASE tools?

The JWST was examined as an example of CASE tool-chain dependency, where all developers used the identical tool chain. This was highly successful in mitigating a similar risk.

The Lockheed-Martin approach of reducing the modeling functions to a subset that would "survive" tool changes was examined but was not tested.

Model exchange is still an issue for tool dependency, not only for systems and software but also for computer-aided design (CAD)/computer-aided manufacturing (CAM) and electronic models.

And, the issue extends to the Unified Markup Language (UML) modeling language. Currently, the Object Management Group (OMG®) is working to improve model exchange definitions, such that the UML standard would specify exactly how models are exchanged and executed across all tools.

So, a risk does exist when mitigation depends on specific tools, and the dependency must have a mitigation plan.

A-1.3 Future Use of CASE Tools

It is recommended that future programs and projects implement a modeling plan that encompasses the following:

- **Integrated tool chain:** use CASE tools that support the model-driven software development approach as part of an integrated tool chain that supports the entire software life cycle.
- **Processes and standards:** establish processes, metrics, and a set of standards, guidelines and best practices for effective model-driven software development.
- **Traceable:** use a tool chain that supports consistency and traceability between different artifacts. Many tools provide automated support to address these issues.
- **Analyzable:** use tools that allow the user to create and maintain models in an analyzable form. Many CASE tools out of the box provide some basic and automated model validation capability. Further, many analysis tools are available for the V&V of such models early in the software life cycle (e.g., model checkers, static analyzers, automated test-case generators).
- **Executable:** Reduce uncertainty and ambiguity by using executable models early on. Many CASE tools provide animation and simulation of early design models.
- **Training:** Train the personnel in the effective use of the tools.

A-2.0 Introduction

CASE tools are maturing to the point where the tools offer a real chance at handling the complexity of software design. However, NASA has little experience with the use and limitations of these CASE tools.

This assessment studied the use of CASE tools within the CxP, specifically, whether CASE tools could mitigate the risks associated with managing the complex interfaces and development of the program software.

A-2.1 Motivation

The complexity of mechanism and structure development now exceeds what was once accomplished by drawing tools and blueprints. CAD/CAM tools have addressed the need for tools that could handle that complexity. Further, the complexity of electronic circuit design now exceeds what was once accomplished with logic boards and dual trace oscilloscopes; the SPICE (Simulation Program with Integrated Circuit Emphasis) simulation software has addressed the need for tools that could handle that complexity.

The complexity of software development is in need of similar tools. Software is still being developed using text editors and compilers. These are implementation tools. Currently, the predominant NASA design tools for software design are text documents, spreadsheets, and slide shows.

A-2.2 Team List

Name	Discipline	Organization
Core Team		
Michael Aguilar	NASA Technical Fellow for Software	GSFC
Peter Berg	Stinger Ghaffarian Technologies, Inc.	ARC
Maddalena Jackson	Software Systems Engineer	JPL
Michael Lowry	Reliable Software Engineering Group	ARC
Masoud Mansouri-Samani	Stinger Ghaffarian Technologies, Inc.	ARC
Christian Neukom	Stinger Ghaffarian Technologies, Inc.	ARC
Thomas Pressburger	Reliable Software Engineering Group	ARC
Nicolas Rouquette	Flight Software Systems Engineering and Architectures	JPL
Marc Sarrel	Ground Systems Engineer	JPL
Kenneth Starr	Engineering Applications Software Engineer	JPL

A-3.0 CxP Tool Usage and Documentation

CxP consisted of numerous hardware, software, and contractual interfaces, creating a complex environment for the development of a complex vehicle. In many cases, L3 development was in progress before the L2 documentation was in place. At L2, the Software and Avionics Integration Office (SAVIO) was compiling an L2 model and document set from the L3 development.

A-3.1 The Unified Modeling Language (UML) Standard

The OMG® is an international, open membership, not-for-profit computer industry standards consortium. Founded in 1989, OMG® standards are driven by vendors, end users, academic institutions, and government agencies. OMG® Task Forces develop enterprise integration standards for a wide range of technologies and an even wider range of industries. The OMG®'s modeling standards, including the UML and Model-Driven Architecture (MDA), enable powerful visual design, execution and maintenance of software, and other processes. UML defines a specific set of models, based on a specific set of graphical notation, used to define the processes, actors, and actions of a system.

UML has gained recognition and general support among CASE tool developers. The CxP tools were generally selected due to their conformance with the UML standard.

A-3.2 CxP Documentation Reviewed

The following documents formed the basis for the model development:

- The L2 documents focused on high-level description and analysis of the systems-of-systems interactions.
  - CxP 70078, Constellation Program Computing Systems Architecture Description Document (CSADD) [ref. 1]
- CxP L3 ICD documents focused on low-level networking and information encoding, packets, and protocols:
  - CxP 70091 [ref. 2]

- CxP 70180 [ref. 3]
- CxP 70187 [ref. 4]
- CxP 70189 [ref. 5]
- CxP 70190 [ref. 6]

A-3.2.1 CxP Documents from WindChill®

Some documentation was managed within the process-based mission assurance Web site communities; other documents were under configuration control within PTC® WindChill®. All were configuration managed at the level of documents, except for requirements that could be managed at the level of individual requirements.

The CAD tools could manage individual mechanical parts. The design tools that were used in circuit-board design could manage individual components.

WindChill's index tracked process-related information (e.g., document numbers) but not software engineering-related items that had identifiers (e.g., commands, telemetry, measurements, functions, packets, etc.). Changes to these software items needed to be maintained using another process.

CASE tools are capable of configuration management and tracking of the "components" within a model through design and implementation. This affords a more detailed management of versions and changes.

A-3.2.2 L2 SAVIO Model (MagicDraw®) and the CSADD

A model using the CASE tool MagicDraw® was developed to produce the CSADD. This model was intended to incorporate and consolidate all of the L3 development that was already in progress.

A-3.3 Technical Issues

Table A-1 shows the problems and suggestions that were offered early in the assessment.

Table A-1. L2 Technical Issues

	Positive Factors	Problems	Suggestions
CxP Documents	Consistency between IRD/ICD & CSADD/C3I Data Exchange Message (DEM)	Bottom-up design ICDs focused on low-level details Duplication across ICDs & IRDs	Define L2 behaviors before defining ICDs Closed-loop state-based behaviors
L2 SAVIO & CSADD	Automation for entering need-line attributes Automation for tables & figures in CSADD	Synchronization between scenarios & activities Inter-view conformance	Scenario generation Modeling phases & configuration changes
OMG specs: UML, SysML	Model Interchange Working Group (MIWG) Fixed OMG SysML 1.2 deliverables & strengthened compliance with applicable OMG specs (XMI, SMSC, QVT)	Type checking support for activity modeling View/Viewpoints Ports/Flows for ICDs	Enter OMG issues to address gap areas (e.g., well-formedness and type checking) Discrete event simulation for system states (QUDV, fUML, SysML, Alf)

The CxP document set maintained consistency between the interface requirements document, the ICD, and the CSADD/C3I model. Definition of the Data Exchange Message (DEM) protocol defined a packet protocol to be used across interfaces. Abstracting and standardizing data exchanges over interfaces reduced the complexity of interface development and configuration management.

L2 SAVIO and CSADD promoted automation and processes to generate attributes, tables, and figures. The CASE tool was used, along with in-house-developed processes, to generate the final document data. Consistency was maintained between the model and the document set, even with the dynamic development that was occurring.

The Model Interchange Working Group worked to improve compliance with the OMG® Systems Modeling Language (SysML) specifications. This work at the specification level is still being pursued with NASA playing a major role in defining the standards. A member of the NESC Software Technical Discipline Team participates in the Specification Management Subcommittee (SMSC). The SMSC is chartered to manage publication and do version management of all adopted standards documents published by the OMG®. The following are specifications and standards that are being improved to support model interchange.

- QVT (Query/View/Transformation) is a standard set of languages for model transformation defined by the OMG®.
- Quantities, Units, Dimensions, Values (QUDV) defines systems of units and quantities for use in system models.
- The XML Metadata Interchange (XMI) is an OMG® standard method for saving UML models in XML. XMI shows how to save any meta-object facility (MOF)-based metamodel in XML.
- Foundational UML (fUML) is an executable subset of standard UML that can be used to define, in an operational style, the structural and behavioral semantics of systems.
- The Action Language for Foundational UML (Alf) is a textual surface representation for UML modeling elements with the primary purpose of acting as the surface notation for specifying executable (fUML) behaviors within an overall graphical UML model.

The CxP efforts focused on consistency between documentation and the exchange of documentation. The focus of this assessment was the mitigation of CxP Risk 4421 through the use of CASE tools. CxP was looking to mitigate the inherently complex software system being developed and the large number of stakeholders by sharing and exchanging models and by developing a model at L2 that drove the L3 requirements. The mitigation goal was to model L2 and exchange the L2 model across development boundaries, both internal and external to NASA.

A-4.0 CxP CASE Tool Experience

A survey was conducted of the CASE tools that were in use by CxP. This included modeling tool experience within Lockheed Martin and the Orion Program and Ares projects.

A-4.1 Use of Commercial-off-the-Shelf (COTS) modeling tools in CxP

As of fiscal year 2010, CxP used three modeling tools from three different vendors:

- MagicDraw®, for the L2 SAVIO model
- IBM® Rational® Rhapsody®, for the L3 Orion model
- Enterprise Architect for the L3 Ares 1 model

A-4.2 Importing L3 Project Models into L2 Program Models to Verify ICD Correctness and Completeness

As will be noted throughout the document, exchanging models across Program interfaces became a sought-after capability to mitigate the complexity addressed in CxP Risk 4421.

For the CxP, the primary documentation at these complex interfaces were ICDs, which captured detail using tools such as Microsoft® Excel®, Microsoft® Word, and Microsoft® PowerPoint®.

In a modeling sense, an ICD is a viewpoint within the model. Different ICDs represent differing viewpoints of the same model. These viewpoints could be generated from the model, either manually or via automation, to create reports and documents, with the model as the single source. This was not the CxP process.

A-4.3 Orion Tool Usage and Documentation

A-4.3.1 Orion Project

Lockheed Martin wholeheartedly embraced the model-driven design and development approach. Within Lockheed Martin, several CASE tools and standards were being integrated into a single tool chain. The tool chain was intended to support the entire software life cycle from analysis, to requirements definition using OOA) and design using OOD, through implementation and testing. The tool chain was intended to provide full traceability between different artifacts.

A-4.3.2 Orion's Original Tool Chain

Lockheed Martin initially selected iUML/iCCG/TA5M from Kennedy Carter Ltd. as the UML-based development tools.

- The CASE tool used a subset of UML called Executable UML (xUML).
- The CASE tool provided iUML: the Development Environment and Simulator.
- The CASE tool provided iCCG: the Configurable Code Generation framework, which provided an extremely adaptable "backend" that could be defined to produce the final source code.
- The CASE tool provided TA-5M: an iCCG-defined C++ auto-coder. The TA-5A source code was provided so that it could be adapted.

Lockheed Martin initially planned to autogenerate 100 percent of the code from the models so that they could be retargeted to different platforms with minimal human intervention.

Lockheed Martin initially planned to embed MATLAB®/Simulink® models into iUML and generate all of the code using iCCG.

Lockheed Martin's original vision was that all integrated code would be generated through iCCG.

A-4.3.2.1 NASA Effort to Support Orion's Original Tool Chain

In fiscal year 2006, Lockheed Martin planned to use the Kennedy Carter iUML. This iUML tool suite comprised a modeler and simulator supporting executable models, plus code generation. The tool supported the OMG® MDA approach to design and development.

Problems with the Kennedy Carter XMI export led to a proactive intervention, where NASA's Reliable Software Engineering (RSE) group at Ames Research Center (ARC) (Exploration Systems Architecture Study (ESAS) project 6G) fixed the Kennedy Carter model exporter to exchange models with IBM® Rational® Software Architect 7 in compliance with OMG® specifications.

- When iUML and iCCG were initially selected as part of the Orion tool chain, it soon became obvious that there were no advanced analysis or V&V tools available for xUML models.
- The RSE group at ARC integrated the Java™ Pathfinder state chart (JPF-SC) model checker with iUML to analyze the xUML state charts.
- The integration was achieved by adapting the Kennedy Carter TA-8 Java™ code generator to translate the xUML state charts to the format required by JPF-SC.
- This work demonstrated how advanced analysis techniques such as model checking can be applied to UML models developed by CASE tools like iUML.
- It also highlighted the issues and problems of trying to translate xUML (or any other) idiosyncratic state chart semantics into other semantics (e.g., JPF-SC semantics).

A-4.3.3 Orion's Tool-Chain Evolution

The Kennedy Carter iUML/iCCG tools were replaced with IBM® Rational® Rhapsody® due to the following issues/restrictions.

- The iUML tool was not as mature as Rhapsody®.
- It did not integrate with other key tools (e.g., configuration management, MATLAB®/Simulink®).
- Configuration management of incremental files was not possible; the entire project had to be managed as a single configurable item in configuration management.
- It did not have a search capability to find model artifacts with the model.
- No capability existed to merge models developed separately, even using the same tool.
- The Action Specification Language (ASL) lacked expressivity. The ASL was the language used within the model to describe behavior. It was not a programming language but could be translated into a standard (C, C++, Ada) language for source-code generation. ASL lacked the expressions usually found in these standard languages.
- xUML had limitations:
    - Had inconsistencies with UML (e.g., state machine extensions).
    - Did not support aggregation or composition.
    - Models (domains, state charts) had a flat structure.
        - A domain could not import or reference other domains (e.g., a design pattern).
        - No hierarchical state machines.
- Existing C++ auto-coder (TA-5M) was not well suited out of the box for flight code.

  - Source code it produced had gross violations of Orion coding standards. Lockheed Martin eventually concluded that they would need to design the auto-coder from scratch.

- Vendor issues:
  - Vendor location in Europe prevented sharing of information.
  - Kennedy Carter was small with a small market share and could not provide sufficient and timely support to Lockheed Martin.
  - Kennedy Carter was unable to fix bugs, incorporate new features, and provide releases on a timely basis.
- Lockheed's lack of experience with Kennedy Carter:
  - It was difficult to find experienced developers to take key positions within the auto-coder adaptation effort at Lockheed Martin.
  - Other Lockheed Martin projects had previously shifted from Kennedy Carter to IBM® Rational® Rhapsody®.
- Lockheed's experience with Rhapsody®:
  - A much more mature tool.
  - Lockheed Martin personnel had more familiarity and history with the tool.
  - Auto-coder already supported the Green Hills® environment.
  - Already integrated with a number of other tools in the Orion tool chain, including MATLAB®/Simulink®, IBM® Rational® DOORS ®, and Synergy.
  - Better support for UML.
  - Supported SysML, which could be used for systems engineering.

Lockheed Martin changed the emphasis from retargeting through auto-coder adaptation to "survivable semantics." This subset of modeling and programming language features was selected to "survive" future tool upgrades and target platform changes over the lifetime of the mission. Table A-2 describes the final tool suite proposed by Lockheed Martin.

Table A-2. Final Lockheed Martin Tool Suite

Rhapsody	Development of analysis and design models and automated code generation
MATLAB, Simulink, Stateflow	
DOORS	Requirements Management and Tracing
Synergy	Configuration Management, Sharing artifacts among different teams
SysML, UML	Standard Modeling Languages
Green Hills MULTI & Integrity	S/W source compilation and the OS: building and running generated code
LDRA	Automated testing of models and code

A-4.3.4 Orion Guidance, Navigation, and Control (GN&C) Success Story with MathWorks®

Twenty years ago, MATLAB®/Simulink® was mainly used as a prototyping tool. Since then it has matured enough to be used as a software development environment with a reasonable auto-coder.

Prior to Orion, the GN&C team at Lockheed Martin and NASA were using numerous modeling and development tools for control engineering. The Orion GN&C team successfully switched to MATLAB®/Simulink® (around 2008/2009).

Even with the code size increase of approximately 40 percent, the benefits of closing the communication gap between control engineers and software engineers for GN&C, along with the increasing speed of computers, made it acceptable.

A-4.4 Ares Project Tool Usage and Documentation

No specific software development tool chain or standards were selected by the whole of the Ares project; each subproject used the tools that they were most comfortable with for the design and development of software models. Reviews were performed on paper and electronic versions of documents containing a variety of diagrammatic representations from different tools without any consistent semantics. Reviews often occurred supported by pictures created using PowerPoint®, Visio®, Dia, MATLAB®/Simulink®, or Enterprise Architect. Different models in design documents under review were often inconsistent. There were no requirements for model simulation or automatic code generation.

Team members were not trained on different diagrammatic representations, which confounded communication. Sharing was a problem: it was not easy for a team to take a model provided by another team and view, modify, or extend it. It was often necessary to manually reenter the model into another tool. For example, Enterprise Architect UML state chart models from a document had to be reentered as PowerPoint® slides because the team did not have access to Enterprise Architect or the model.

A-4.4.1 Example Ares Subprojects

A-4.4.1.1 Vehicle Systems Management Project

The project team used Enterprise Architect (a UML-based tool).

Design documents included the following UML representations:

- Use case diagrams
- Class diagrams
- Interface diagrams
- State machine diagrams
- Activity diagrams
- Sequence diagrams

No plans existed for simulating models or generating code from those models.

A-4.4.1.2 The GN&C Project

The project team used MATLAB®/Simulink® for analysis and prototyping.

The logic in the design was specified through a set of flowcharts and equations associated with each of the steps identified in the flowcharts.

Design documents included the following non-UML representations created using PowerPoint®:

- Process flow diagrams
- Flowchart diagrams
- Architecture diagrams

A-4.4.1.3 The Ares FDNR (Fault Detection, Notification, and Response) Project

Early on, Ares selected Enterprise Architect as the standard CASE tool. The majority of the personnel either did not acquire the software or receive any UML training. No single formal language (such as UML or SysML) was used in developing the SDD.

Design documentation included a mixture of the following diagrams:

- UML diagrams (Enterprise Architect):
  - Domain view diagrams
  - Use case diagrams
  - Activity diagrams
  - Sequence diagrams
- Non-UML diagrams (Dia, PowerPoint®, Visio®):
  - Logic diagrams (Dia)
  - Flowchart diagrams (PowerPoint®, Visio®)
  - Architecture diagrams (PowerPoint®)
  - Freestyle (Visio® or PowerPoint®)
  - Functional flow (PowerPoint®)
  - Process flow (PowerPoint®)

A-4.4.1.4 The Ares Modeling Effort at ARC

The team at ARC used MATLAB®/Simulink® to model the abort algorithms, used other MathWorks® tools (e.g., SystemTest™, Simulink® V&V toolbox) to verify the algorithms, and used Stateflow® to model the mission timeline as well as the abort-related interactions between the Orion and Ares models and to integrate and simulate these models together.

The ARC team manually translated the UML state chart diagrams contained in the Vehicle Management System SDD into Stateflow® for the purpose of analysis using the V&V tools from MathWorks®.

The ARC team used IBM® Rational® DOORS® for requirements management.

A-4.5 Survey of CxP CASE Tool Capabilities

Tool capabilities used at L2 and L3 to capture and model the L2 interfaces were surveyed to determine general capabilities. Not all capabilities were used during CxP (see Table A-3).

Table A-3. Survey of CxP CASE Tool Capabilities

Features	Enterprise Architect	Rhapsody	MATLAB/Simulink
Analysis and design support	Yes. Using UML, version 2.1	Yes. Using UML 2.0	Yes. Using Simulink and StateFlow
Systems Engineering Support	Yes. Using SysML	Yes. Using SysML	Yes.
Model Validation	Yes. Default set available. Also user defined possible through OCL.	Yes. Default set available. Also user defined through an API	Yes. Default set available.
Model Simulation	No	Yes	Yes
Code Generation	Yes. C, C++, Java, C#, Ada, Visual Basic, Python, others…	Yes. C, C++, Java, Ada	Yes. C, C++
Testing	Yes. Support (JUnit, NUnit)	Rhapsody TestConductor add-on available for automated testing. Allows CppUnit and JUnit test integration.	Automated testing, model-checking, test coverage tools available (e.g., SystemTest, Design Verifier, Simulink V&V Toolbox)
Requirements management	Models, manages and traces requirements. Integrates with DOORS	Models, manages and traces requirements. Integrates with DOORS, Word, Excel, etc.	Integrates with Word, Excel and DOORS
Model import/export support	Supported through XMI.	Supported through XMI.	Import of Fortran, C and C++. Export of Simulink models to Rhapsody as a object in an object diagram
Maintaining consistency of multiple diagrams	Yes.	Yes.	Yes.
Document Generation	Yes	Yes	Yes
End-to-End Traceability	Yes	Yes	Yes
Reverse Engineering	Yes	Yes	No.

A-4.6 Problems with UML tools

A-4.6.1 Incompatibility with Standards

Most UML tool vendors claim full compliance with the UML standard, but there are many inconsistencies with the formal standard. Minor inconsistencies might be in the graphic visualization for some UML elements. Major inconsistencies might affect actual model behavior, such as specifying an event trigger on an outgoing transition from an initial vertex.

Many of these violations often can be checked automatically through built-in model validation. In some cases, models that contain such violations can be rewritten or automatically transformed to remove these inconsistencies.

A-4.6.2 UML Semantics Variations

The UML specification semantics are intentionally underspecified to provide leeway for domain-specific refinements of the general UML semantics. This allows UML to become a family of languages that can be used for modeling different application domains, and enables specialization of those parts of UML for particular situations or domains. Each tool vendor chooses their own interpretation of these semantics.

These interpretations within the UML semantics can affect the behavioral model such that the same behavioral model (e.g., a state chart) developed on different tools may run quite differently when executed. The UML specification was not designed rigidly enough for execution consistency.

A-4.6.3 Model Import/Export

XMI is a standardized format for model interchange between UML-based tools, but in practice an XMI file exported from one tool is unlikely to import correctly into a different tool. Specialized translation tools can bridge between the different XMI dialects. This makes model sharing difficult.

No verification or validation is available, nor guarantee of quality for the import and export of models between different vendor tools.

A-4.7 The JWST effort at Goddard Space Flight Center (GSFC)

Independent of the CxP, the development team at GSFC has designed, developed, and tool generated the flight code with the IBM® Rational Rose® RealTime UML real-time development tool suite. The Integrated Science Instrument Module (ISIM) integrates all of the hardware and software to support the JWST science instruments. The core command and data handling (C&DH) software was developed using the CASE tool, as were all of the applications controlling the specific science instruments.

The Science Instrument (SI) applications were developed by several organizations at their facilities and following their own review processes (see Table A-4).

The use of a CASE tool for development was primarily to reduce the complexity of integrating the independently developed application software.

Table A-4. JWST Software Application Developers

- **GSFC**
  - Management, Systems Engineering, & Science
  - ISIM C&DH Electronics/Software, including Remote Services Unit
  - ISIM Structure, IEC, and Thermal Control Subsystem
  - ISIM-level I&T
  - NIRSpec Micro-Shutter and Detector Subsystems
- **European Space Agency/European Consortium**
  - NIRSpec Instrument
  - MIRI OBA (Optical Bench Assembly)
- **Canadian Space Agency**
  - FGS/TF
- **University of Arizona with Lockheed Martin**
  - NIRCam Instrument
- **JPL**
  - MIRI Management & Systems Engineering
  - MIRI Detector Subsystem, Cooler, and Flight Software
- **JWST Prime Contractor Team (NGST, Ball, ITT, ATK)**
  - ISIM Thermal Management System (enclosure) & GSE for ISIM I&T
- **Space Telescope Science Institute**
  - On-Board Scripts and Flight/Science Operations

A-4.7.1 Enhancing Systems Integration through Standardized Development Resources

The ISIM core C&DH flight software (FSW) architecture, design, and requirements are based on the GSFC Flight Software Development Branch's heritage C&DH architecture that traces back to the Solar, Anomalous, and Magnetospheric Particle Explorer (SAMPEX) mission launched in 1992. Even though less than ten percent of the actual heritage FSW was reused due to the JWST ISIM FSW being implemented in the UML paradigm, key development team members' significant experience with the heritage FSW architecture allowed them capture the essence of heritage design to a high extent.

To deal with the International Traffic in Arms Regulations (ITAR)-related issues, the core C&DH services were built and delivered to the SI development teams as an ITAR-compliant library, source-code free interface definitions, and generic application integration environments. To deal with potential integration issues early in the development process, NASA provided a development platform based on the Rational Rose® development tool suite, an SI application design and coding template of generic interfaces, a common ground system, and a COTS runtime target environment that provided all of the functionality of the ISIM onboard processor environment. These NASA-provided SI FSW development, test and integration, and validation assets were identified as the Science Instrument Development Unit (SIDU).

A-4.7.2 Common Development Tools and Methodology

To facilitate integration and to help ensure consistent design implementation, all FSW development teams were required to use a common set of tools, language, real-time operating system environment, and development methodology. The teams were required to use C/C++, UML, and VxWorks® for development. NASA provided a design guidelines document with which all teams were required to comply.

A-4.7.2.1 Development Tools

The Rational Rose® RealTime UML real-time development tool was used for software design, coding, testing, and integration. The ISIM FSW development team is one of the few organizations in the high reliability space flight environment to actually use the use auto-source-code generation aspects of the Rational Rose® RealTime tool, as many organizations were limited to the Rational Rose® diagramming and modeling features. The ISIM FSW development team coded within the UML design model elements with the objective of ensuring that the code and design were always in sync, and significantly increased productivity as design and code reviews did not require resource-intensive review material preparation. The Software Documentation Automation (SoDA) tool and provided Word templates were used for automated creation of code and design review documentation using model elements and database contents. The IBM® Rational® RequisitePro® software was used for requirements management. RequisitePro® was part of the integrated tool suite that provided links to the configuration management and defect tracking tools and enabled traceability down to the source code implantation levels. IBM® Rational® ClearQuest® was used for change management and defect tracking. IBM® Rational® ClearCase® was used for version control and change tracking. The tool helped facilitate concurrent development by the multideveloper ISIM FSW team by permitting the check-in/check-out of any computer software configuration item or lower level software component as individually controllable elements.

The ISIM ground system support team delivered over 21 COTS-based development, test, and integration systems. These systems are being used to validate and qualify flight instrument hardware for space. All of the systems had an identical development environment. All of the systems came with the Rational Rose® RealTime UML real-time development tool suite.

A-4.7.3 CASE-tool-generated Code Build Verification

Concern over CASE-tool-generated code has always been an issue. The code generated by Rational Rose® RealTime was reviewed and found to be acceptable for the mission. The generated code, as C source files (see Figure A.1), was compiled and passed all unit, subsystem, and system testing, indistinguishable from hand-written code. The code was submitted for testing by industry standard static code checkers. (Static code checkers perform source code analyses to ensure that specific standards are being followed, to determine whether coding language issues have been addressed, and to identify coding errors.)

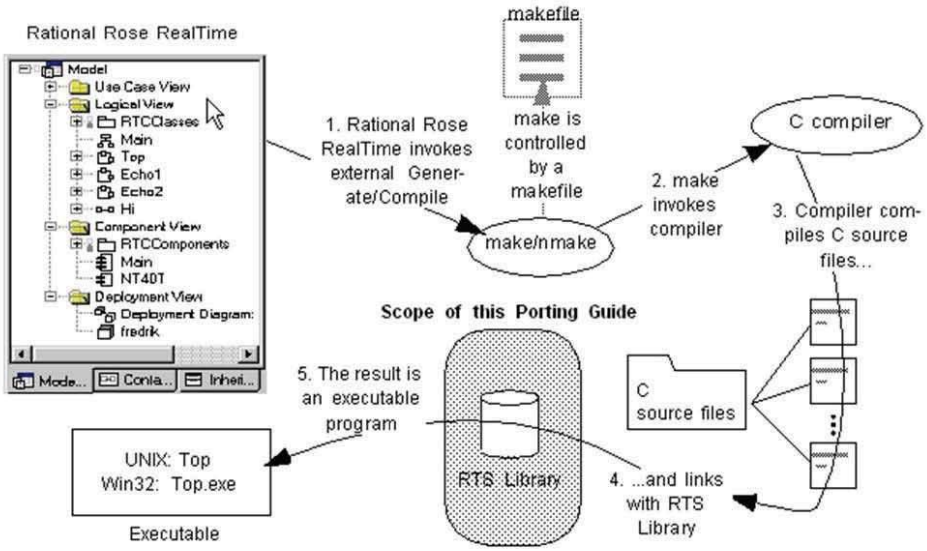


Figure A-1. Rational Rose® Code Generation

A-4.7.3.1 Reasoning Static Code Analysis

The GSFC Flight Software Systems Branch management sponsored an independent analysis of the ISIM FSW by the Reasoning Static Code Analysis Company. A total of 1,046 files containing 140,923 lines of C++ source code were analyzed using the Reasoning Static Analysis tool. Eleven total defects were found: an amazingly small number for the size of the code and the current phase of development. The breakdown of the 11 defects was:

- Two out-of-bounds array accesses
- Four uninitialized variables
- Five false positives

A-4.7.3.2 Coverity Static Code Analysis

The NESC's Software Discipline Expert sponsored an independent static code analysis of the JWST source code by Coverity, Inc., in September 2006. Results of the Coverity analysis indicated:

- The JWST had no severity-level-1 or -level-2 errors.
- The defect density was extremely low.
- The total number of defects identified was lower than for other programs evaluated that had significantly fewer lines of source code.

A-4.7.4 ISIM Project Level Integration and Testing

The SIDU systems are being used to validate and qualify flight instrument hardware for space.

The Fine Guidance Sensor (FGS) team in Canada has a unique FGS validation test system that allows them to validate their tight fine-guiding timing requirements. The integrated C&DH hardware team is using the ISIM FSW to qualify flight hardware for space through formal box level environmental testing. The ISIM system-level integration and testing is using the ISIM FSW for test-procedure development and operator training. The Space Telescope Institute is using the ISIM FSW to develop and certify operational scripts for space flight use.

The initial full integration for all science application software was accomplished in one week. Some individual instruments were integrated in a single day.

Similar to the CxP, the JWST ISIM project had complexity issues across interfaces. Specific to the flight software, these interfaces spanned the globe. The size (see Figure A-2) and the operating environment of the telescope complicated the issue. The instruments were designed to operate at below 40 degrees Kelvin. Many of the instrument mechanisms would not operate above cryogenic temperatures. Subsystem testing was required to be performed at the facilities where the hardware could be verified at these temperatures.

Figure A-2. JWST Full Size Mockup

A-4.7.5 JWST ISIM Software Interface Complexity Mitigation

By requiring identical development tools and environments capable of emulating the entire ISIM software, the complexity of different interfaces was reduced. Identical development tools enhanced communication between and among the developers. Documentation for the system was similarly generated. Further, ITAR issues were mitigated by allowing the common model to be exposed only at the interfaces. The deeper model was not viewable, and the code was shared among the developers as a compiled library.

In essence, the science instrument applications were integrated with the common flight code at each development site. Final integration of all of the science applications with the common code on the flight hardware was the last outstanding integration effort.

Key complexity mitigation points were as follows:

- External software developers were supplied with identical tool suite and hardware development environment. The Rational Rose® tool suite, COTS C&DH hardware, ground system, and database tools were supplied.
- Complete C&DH model was supplied to NASA developers.
- Library of C&DH model was supplied to ITAR-restricted developers.
- Integration of the C&DH model and instrument application-specific model occurred for all development teams at the model level.
- Training, lessons learned, support, and guidance were shared across all development teams due to identical environments and development tools.
- Review presentation material was similar in content across all external and internal reviews.

A-5.0 Integrated Modeling

The following integrated modeling examples demonstrate executable models of the Ares and Orion projects. One model example demonstrates an executable model of the mission timeline with ground activities, vehicle activities, ground systems, vehicle systems, and vehicle software states. A second model example demonstrates the abort communication between Ares and Orion.

A-5.1 Integration of Vehicle System Manager and Mission Timeline

A-5.1.1 Vehicle System Manager and Mission Timeline Model Scope

The simplified diagram of the CxP interfaces is presented in Figure A-3. The scope of this model is the behavior of the portions in blue. The model will consolidate the interfaces into five high-level partitions: Countdown, Mission Timeline, Activities, States, and Vehicle. Each partition contains numerous concurrent state machines. For example, the partition state contains three major state machines: one for Orion systems, one for ground systems, and one for vehicle systems. Then, within the Orion systems, numerous state machines will describe the next lowest level of detail. This continues until the level of detail is sufficient for the problem analysis.

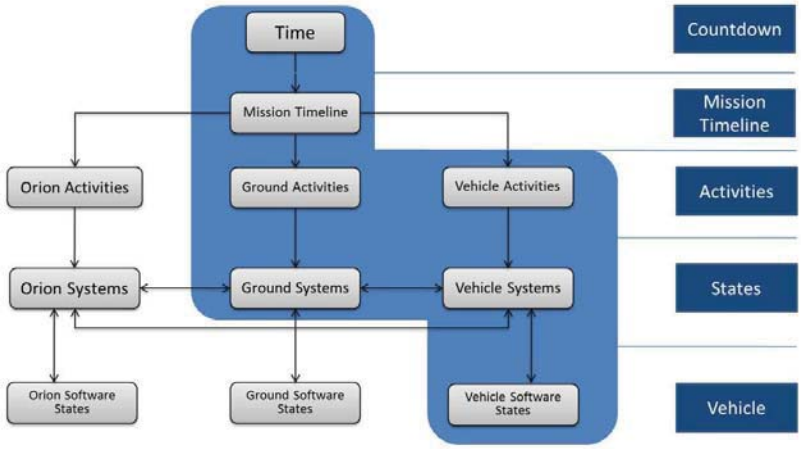


Figure A-3. Scope of Integrated Model of VSM and Timeline

A-5.1.2 Model from Countdown Master Timeline

The Countdown Master Timeline was maintained within an Excel® document. Each entry was modeled as a separate concurrent state machine (see Figure A-4).

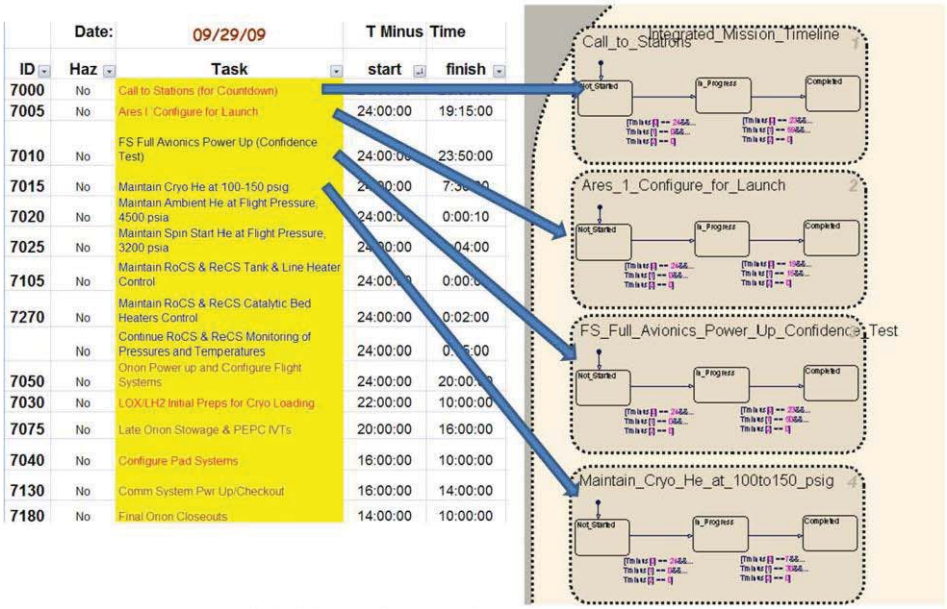


Figure A-4. Translation from Countdown to Executable Model

Modeling tools can describe state machines that run concurrently. Each timeline entry was modeled as an independent state machine to allow for the overlapping and concurrent procedures. Each procedure was modeled using a simple template. Each timeline procedure consisted of three states: Not_Started, In_Progress, and Completed. A start time, given in hours:minutes:seconds, is the condition that allows the transition from the Not_Started to the In_Progress state. An end time, given in hours:minutes:seconds, is the condition that allows the transition from the In_Progress state to the Completed state.

In Figure A-5, Maintain_Cryo_He_at_100to150_psig transitions to In_Progress at 24:00:00 and then transitions to Completed at 07:30:00.

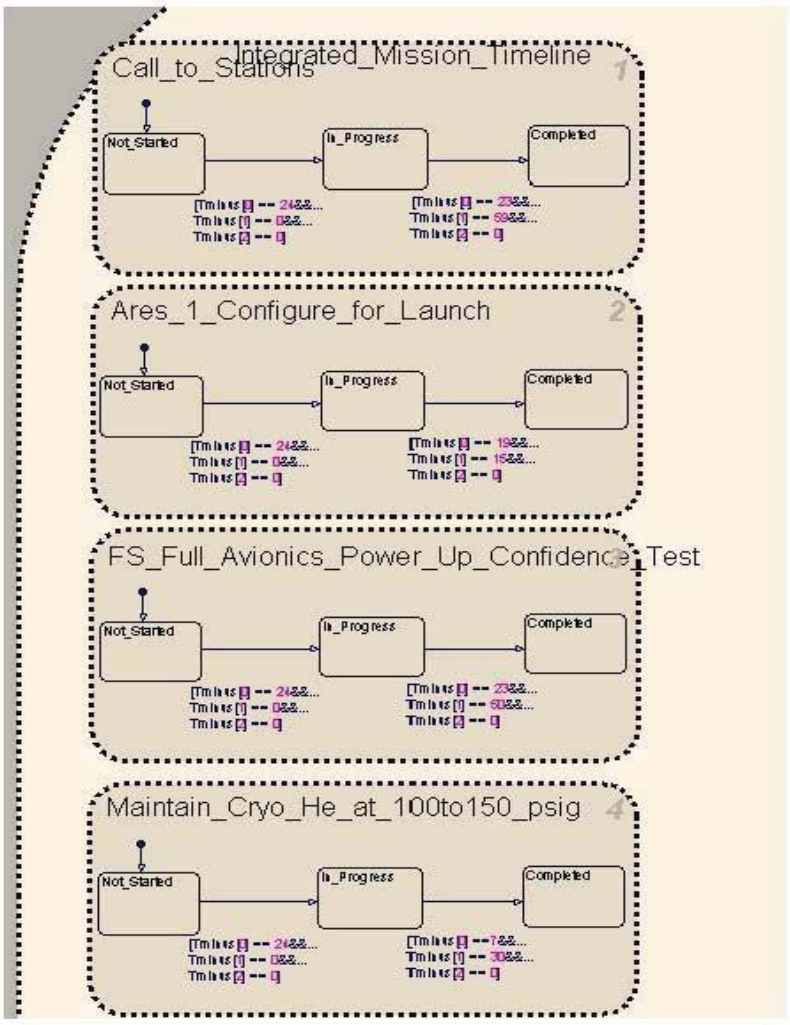


Figure A-5. Timeline Concurrent State Machines

Modeling tools can describe models that are hierarchical. All of the timeline procedures exist within the state machine Integrated_Mission_Timeline. Each of the state machines has an In_Progress state. Modeling tools allow the developer to "dive deeper" into detail by selecting and opening each In_Progress state to develop the detail within the state. An example of this hierarchical capability is presented next.

A-5.1.3 Model of the Vehicle System Manager

The model of the Vehicle System Manager (VSM) was maintained using the Enterprise Architect UML tool. The Enterprise Architect modeling tool was not available to the assessment

team. The decision was made to use the Enterprise Architect model documentation as a source and translated into an executable model using MathWorks® Stateflow® because the team already had expertise in the MathWorks® tools. The executable models afforded demonstration of more types of analysis.

Enterprise Architect UML models of the VSM (i.e., activity diagrams, state charts, and use cases) were modeled in Stateflow® to produce an executable model (see Figure A-6).

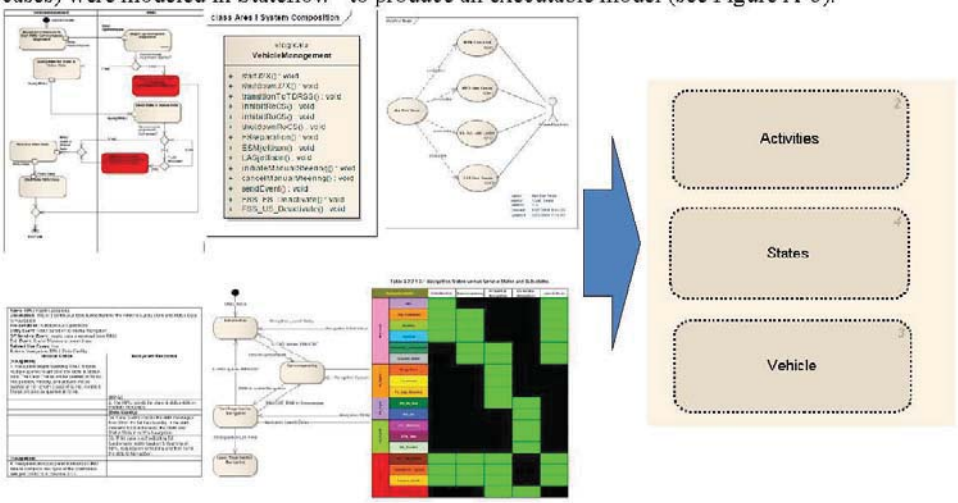


Figure A-6. Translation from VSM to Executable Model

A-5.1.4 Integration of Vehicle System Manager and Mission Timeline

The Mission Timeline and the VSM procedures were integrated into a single executable model. A trace of a portion of the model is presented next.

A-5.1.4.1 Execution of Trace through the Integrated Model

An execution of the model runs an animation based on the external inputs that cause state transitions to occur. The external events can be a script of conditions and events that stimulate the model.

In the following example, time is used as the external input. The countdown timer drives the timeline, and the transitions within the timeline drive all other model transitions. The current transitions and states in the execution are highlighted in blue (see Figure A-7). The countdown timer also can be modeled to generate the time, or time can be stepped manually. The countdown begins at 24:00:00.

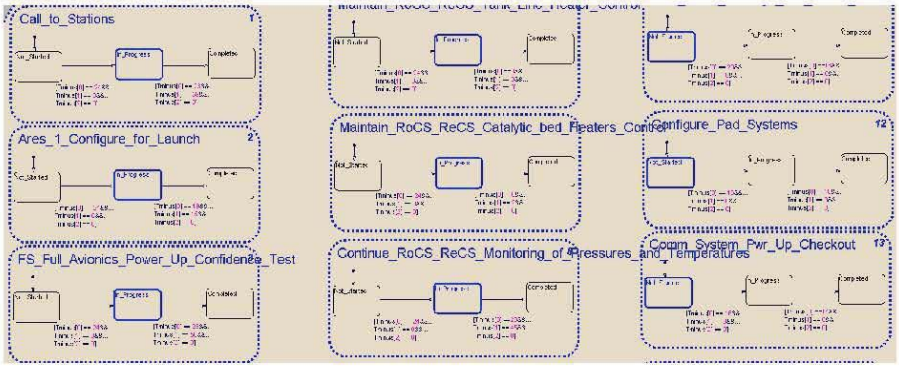


Figure A-7. Initial Execution State of Timeline

Integrated_Mission_Timeline initiates at 24:00:00. In the model, concurrent processes enter the Not_Started state or the In_Progress state. As the countdown continues, the states transition within each process depending on the countdown time, other states, and other transitions.

State transitions within the Integrated_Mission_Timeline cause transitions within the Activities model. The Integrated_Mission_Timeline model contains a process Ares_1_Configure_for_Launch, and Ares_1_Configure_for_Launch is in the In_Progress state at 24:00:00. Concurrently, within the Activities model, the Idle state transitions to the Launch activity (see Figure A-8).

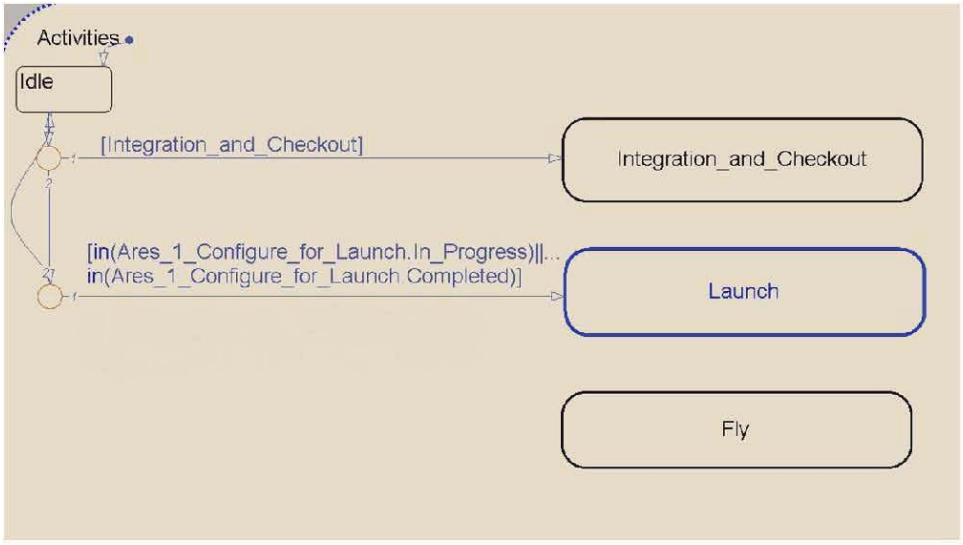


Figure A-8. Activities: Launch State

Using the tool to open the Launch activity displays the details of the model within Launch. The model consists of a single Dry_Checkout state (see Figure A-9).

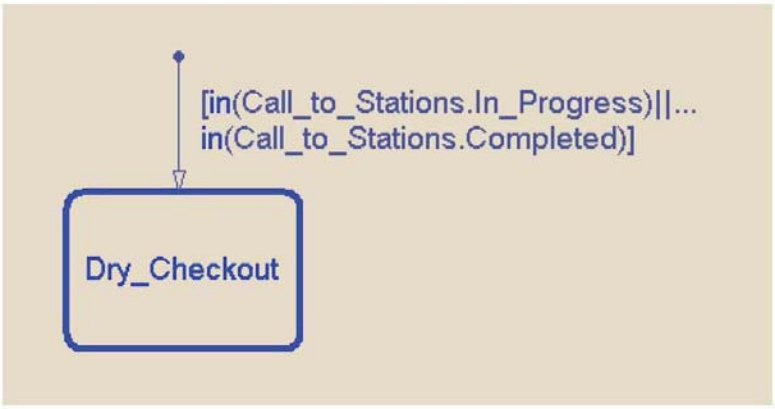


Figure A-9. Launch: Dry_Checkout State

Using the tool to open the Dry_Checkout activity displays the details of the model within Dry_Checkout (see Figure A-10). The model consists of three states, starting in the Powerup state.

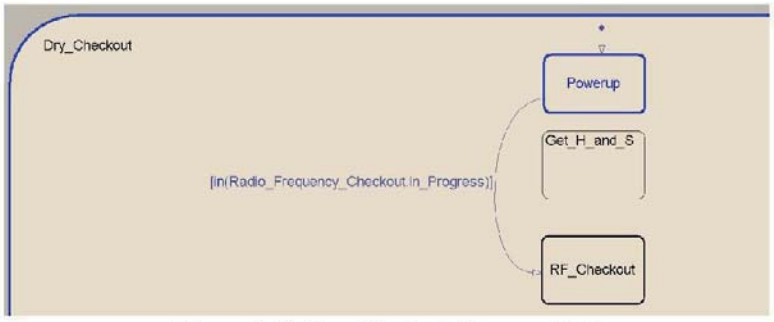


Figure A-10. Dry_Checkout: Powerup State

Using the tool to open the Powerup activity displays the details of the model within Powerup (see Figure A-11). The model consists of several states, with the initial transition into the Ground_Power_On_EPS_Device state.

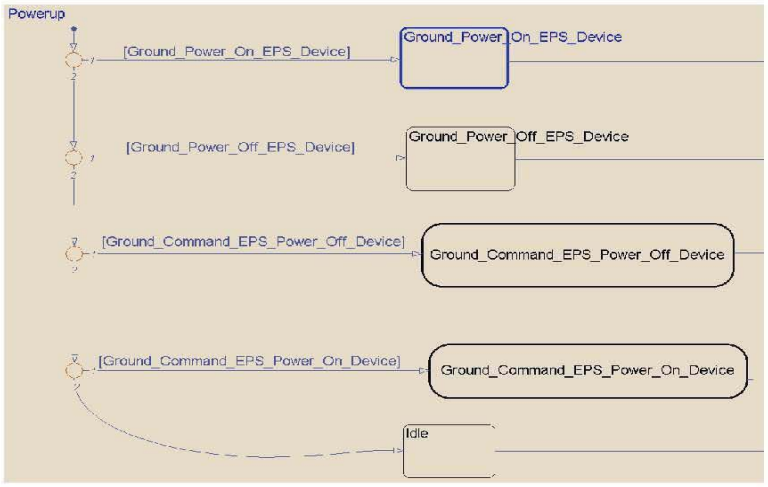


Figure A-11. Powerup: Ground_Power_On_EPS_Device State

Using the tool to open the concurrent state machine model GroundSystems, the tool displays the details of the model within GroundSystems. The model consists of several states, with the current state transition into the Ground_Applies_Power_to_Umbilical_Device state (see Figure A-12).

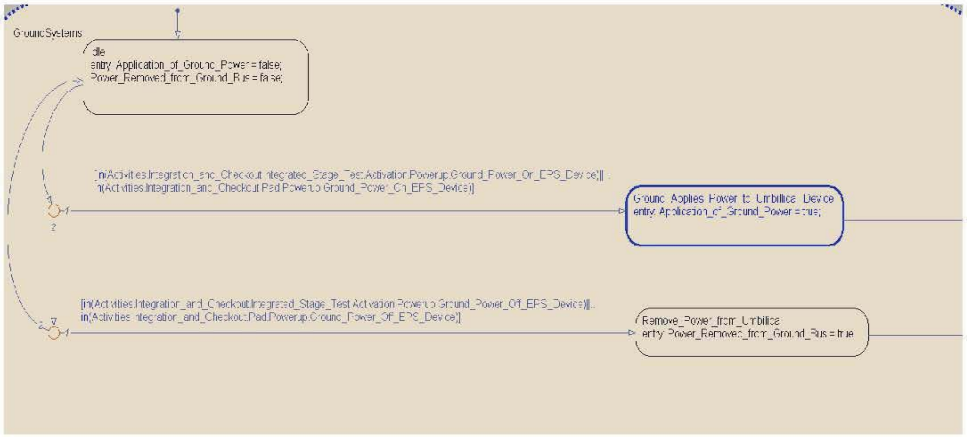


Figure A-12. GroundSystems: Ground_Applies_Power_to_Umbilical_Device State

Using the tool to open the concurrent state machine model EPS_HW, the tool displays the details of the model within EPS_HW (see Figure A-13). The model consists of several states, with the current state transition into the EPS_Device_Switches_on_Normally_On_Loads state. When the ground system applies power to the umbilical device, transitions occur through several states powering several devices. The states EPS_Device_Powers_Up and Device_Performs_BIT were entered and exited. Within the EPS_Device_Powers_Up state, the logic variables

Power_applied_to_PDCU_1_2 was set to true, PDCU 1 was set to "on," PDCU 2 was set to "on," and the data collection units (DCUs) were set to "on."

The EPS_Device_Switches_on_Normally_On_Loads state is a good example of the internal state processing. Each state simply assigns Boolean true or false values to system variables identified in the original Enterprise Architect model. These variables can be traced back from the models to the original requirements documentation.

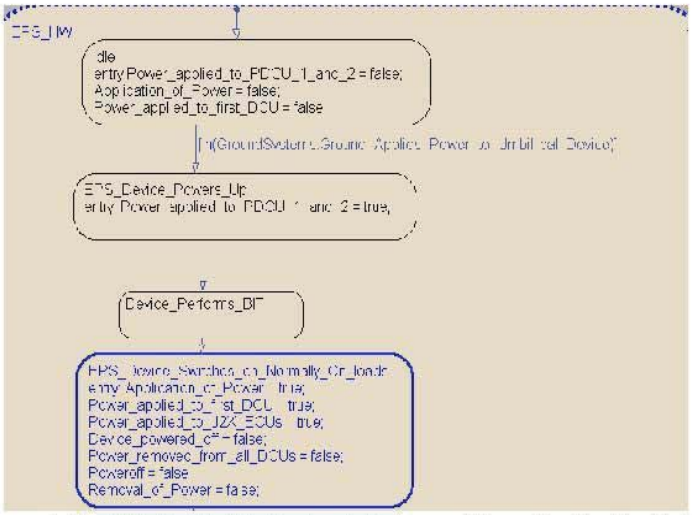


Figure A-13. EPS_HW: EPS_Device_Switches_on_Normally_On_Loads State

Using the tool to open the concurrent state machine model Vehicle (see Figure A-14), the tool displays the details of the model within Vehicle. The model consists of several states, with the current state transition into the Prelaunch state resulting from power being applied to both PCDU 1 and 2.

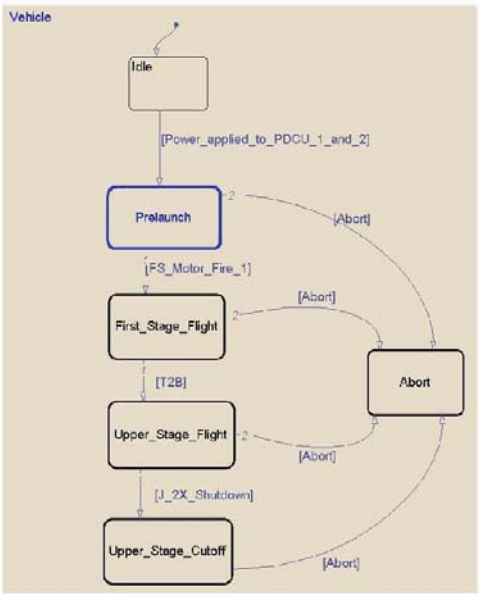


Figure A-14. Vehicle: Prelaunch State

Using the tool to open the concurrent state machine model US_VTC, the tool displays the details of the model within US_VTC (see Figure A-15). US_VTC is a state machine model representing software states. The model consists of several states, with the current state transition into the PowerUp state resulting from power being applied to the first DCU; the Vehicle state machine is in the Prelaunch Idle state.

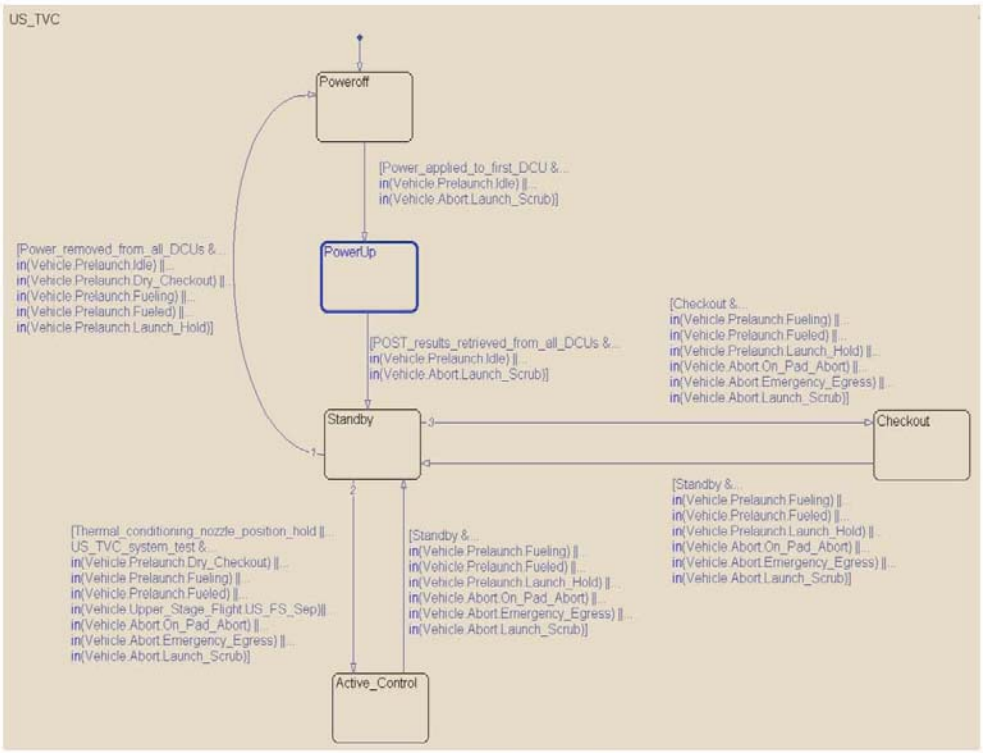


Figure A-15. US_TVC: PowerUp State

A-5.1.4.2 Integration of Vehicle System Manager and Mission Timeline Summary

The above model example describes general CASE tool capabilities. The problem can be partitioned into simple state machines and solved to the degree of depth required. The example consisted of Boolean variables, but other variables could have been modeled. For example, a model of amperage could have been added by simply summing each load into an arithmetic variable as it was powered on. A model of bus communication loading could sum bus traffic for each state.

It is common practice to verify and validate interfaces developed across some institutional or contractual boundary by peer review of written or illustrated documentation. A simple model of the interface developed from the existing documentation sets can expose assumptions, errors, inconsistences, gaps, and exclusions within the documentation sets. The development of the model requires numerous "Is this what you mean?" discussions.

It should be pointed out that the intent of these CASE tools is to provide an environment where the behavior of a system can be discussed and documented during design, much as a "brainstorming" environment far in advance of documentation. Then, once the model has been verified and validated by peer reviews, written documentation can be generated by the tool.

Changes to the model can follow the same process of design, verification, validation, and document generation.

Another capability of these CASE tools is a higher level of detail in configuration management. Most configuration management processes treat an entire document as one configurable item. Changes occur at the document level. Much like CAD tools, CASE tools can treat every individual item within the model as a configurable item (e.g., event, condition, transition, state) such that any change to any item within the model can be tracked and controlled.

A-5.1.4.3 Using Modeling Tools to Catch Integration Errors

There is a formal basis for these CASE tool models. There can be consistency checks and completeness checks of a model by the tool itself. The tool can report these model errors to indicate incomplete, inconsistent, and missing parts of the model.

A-5.1.4.3.1 Using the Tools for Static Model Checking

Each of the following illustrations includes model errors. Models might be in an intermediate state, have states or transitions that need to be determined, or have an actual flaw. Because the CASE tool model has a formal basis, the tool itself can identify incorrect or missing portions of the model.

A visual inspection reveals the Upper Stage Inertial Navigation state cannot be entered, and the Gyrocompassing state will immediately transition to the Launch Delay state under all conditions (see Figure A-16).

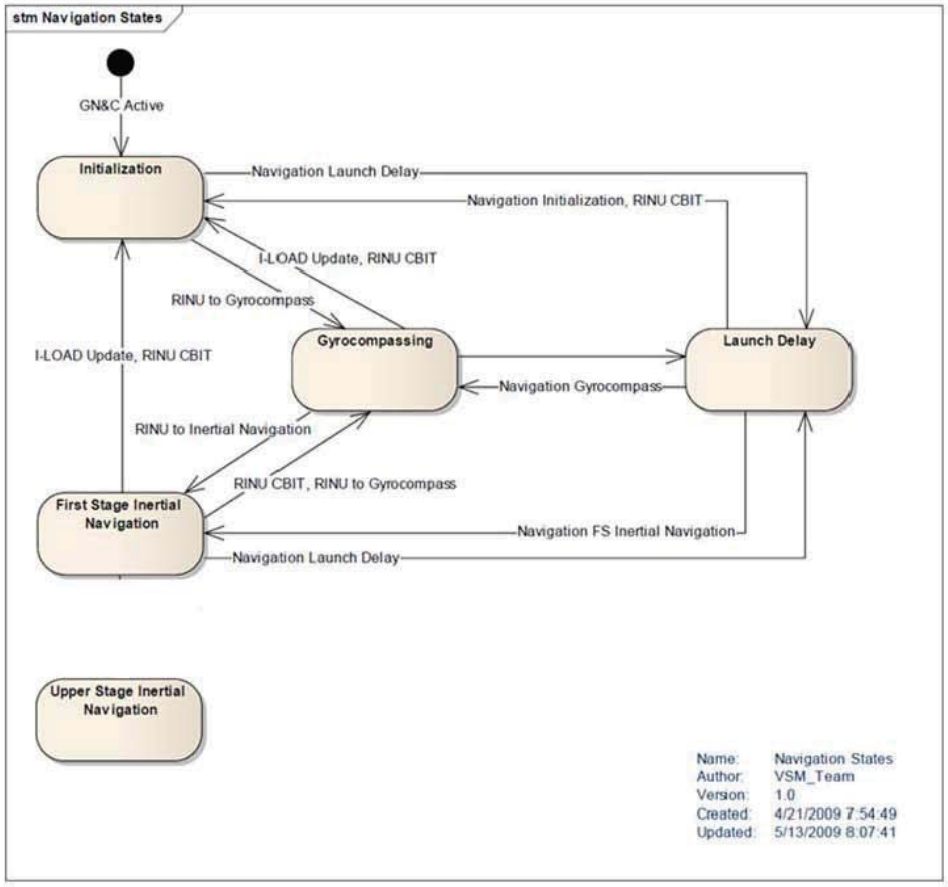


Figure A-16. Graphical Model Errors

A-5.1.4.4 Errors Found Through the Modeling Tool

The CASE tool can produce a report of the model completeness and consistency, as in the example given in Figure A-17. Each of the numbers identifies the component to be reviewed.

```
Runtime error: State Inconsistency
Block Name: VSM/Integrated (#1194 (0:0:0))
Active cluster state has no active substates
State Control (#1331 (0:0:0))
Active cluster state has no active substates
State FSS (#1429 (0:0:0))
Active cluster state has no active substates
State Guidance (#1309 (0:0:0))
Active cluster state has no active substates
State MPS (#1389 (0:0:0))
Active cluster state has no active substates
State Navigation (#1317 (0:0:0))
Active cluster state has no active substates
State Steering (#1370 (0:0:0))
```

Figure A-17. CASE-tool-generated Model Error Report

A review of the report could produce the critique of the model shown in Figure A-18.

Vehicle
Diagram does not have any active states prior to "Power applied to FDCU 1 and 2"

Vehicle.Prelaunch
All returns from Launch_Hold are TBD

Vehicle.Abort
No default state. Suggest adding default state as "Abort_Standby"

FSS, MPS, Guidance, Navigation, Control, and Steering
No default state, suggest adding "Standby" or "Idle"

Navigation
After Gyrocompassing, the Navigation will automatically enter the Launch_Delay state. If this is unintended, suggest placing guard to prevent this

RINU and RGA
Both RINU and RGA use "PowerApplied", "OffCommand", "PerformCBIT", and "gyrocompass" to transition between states.

FSS, US_TVC,
Unclear as to which are OR conditions and which are AND

Figure A-18. Peer Review of Model Error Report

A-5.2 Modeling of Ares-Orion Interface Communication during an Abort

A-5.2.1 Model of Orion-Ares I Communication

The following document was used as the basis for a communication protocol model of the communication between Ares and Orion (see Figure A-19). From this document, three concurrent models were developed. One model captured the Ares behavior, one model captured the Orion behavior, and one model allowed command inputs to be scripted. These command inputs could produce both nominal and off-nominal traces through the model.

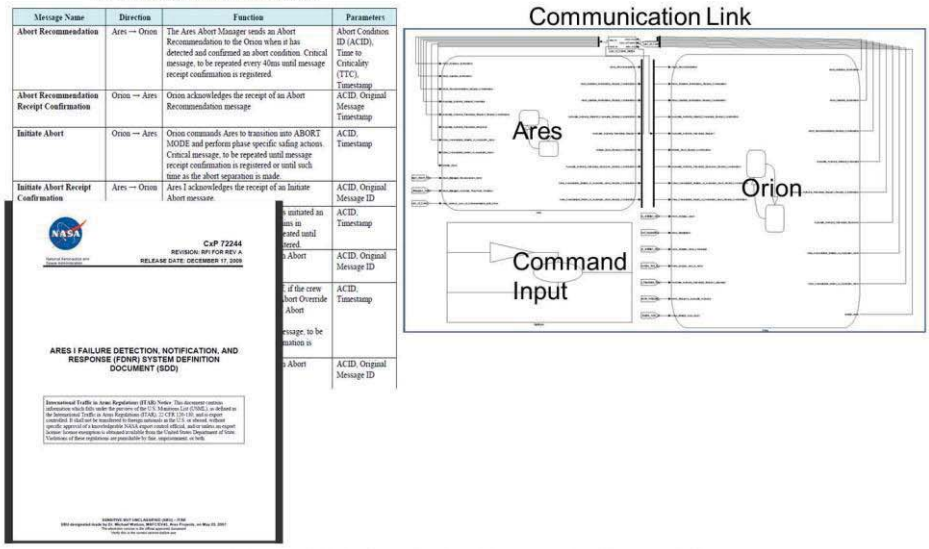


Figure A-19. Ares-Orion Communication Model

A-5.2.1.1 Conversion from Tabular Form to Executable Stateflow® Example

For each entry in the documented table, a model was developed for the protocol. Ares to Orion communication caused Orion state transitions, while Orion to Ares communication caused Ares state transitions. Variables were chosen to model all dependent variables documented in the table (see Figure A-20).

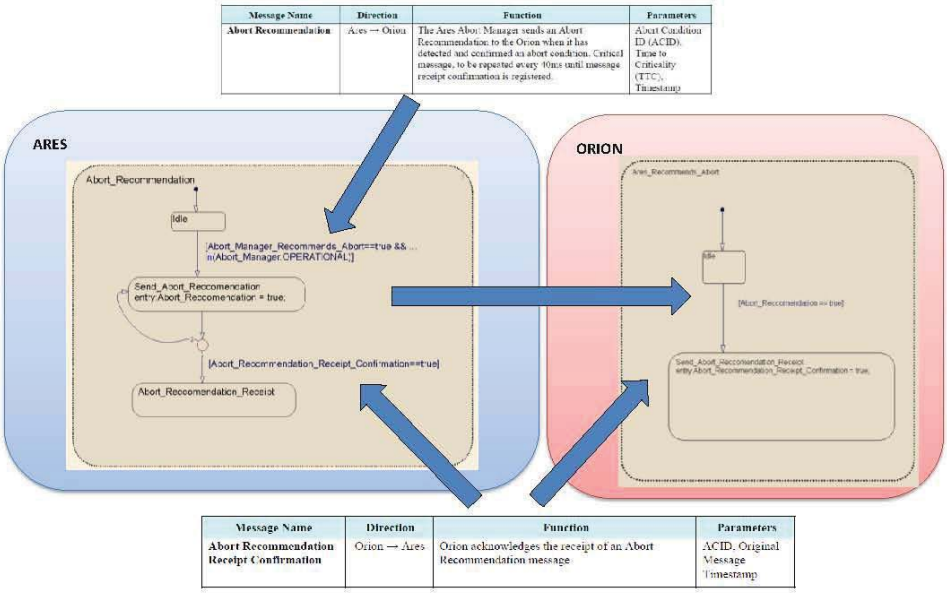


Figure A-20. Communication Table Documentation to Executable Model

A-5.2.2 Trace through "Nominal" Abort Interaction

When this model was completed, the model could be executed under various conditions. The following execution traces a nominal abort execution (Figures A-21 through A-27).

Figure A-21 represents the initial state of the model. The active states are indicated in blue. Note the numerous concurrent state machines modeled for both the Ares model and the Orion model.

A-5.2.2.1 Pre-Abort Recommendation

- **Ares:** All command states Idle except for Abort_Manager, which is in the Operational state.
- **Orion:** All command states Idle except for Crew_Enable_Auto_Abort, which is in the Enabled state.

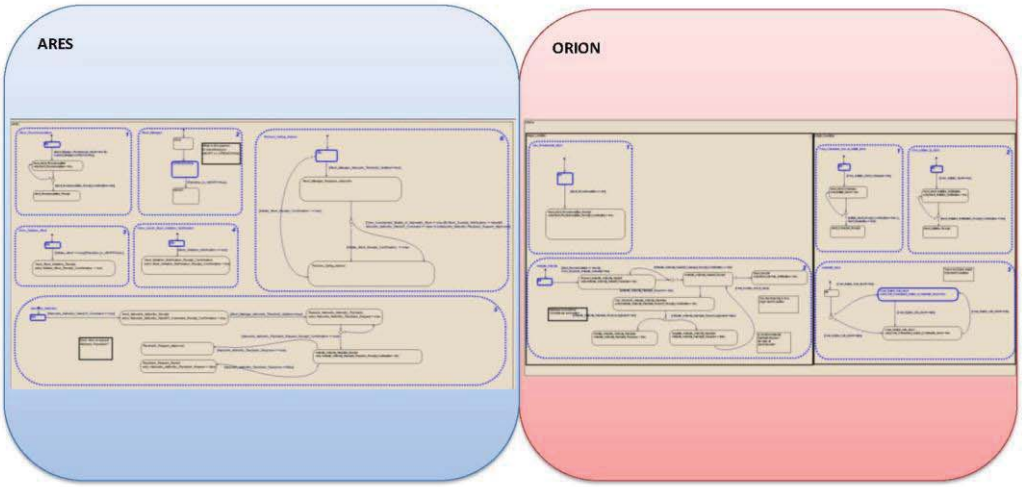


Figure A-21. Pre-Abort Recommendation

A-5.2.2.2 Ares Sends Abort Recommendation

- **Ares:** Abort_Recommendation command state transitions from Idle to Send_Abort_Recommendation and waits for Orion response. All other command states remain Idle.
- **Orion:** Ares recommends abort command state transitions from Idle to Send_Abort_Recommendation_Receipt upon receipt of abort recommendation.
- **Ares:** The abort recommendation state transitions from Send_Abort_Recommendation to Abort_Recommendation_Receipt.

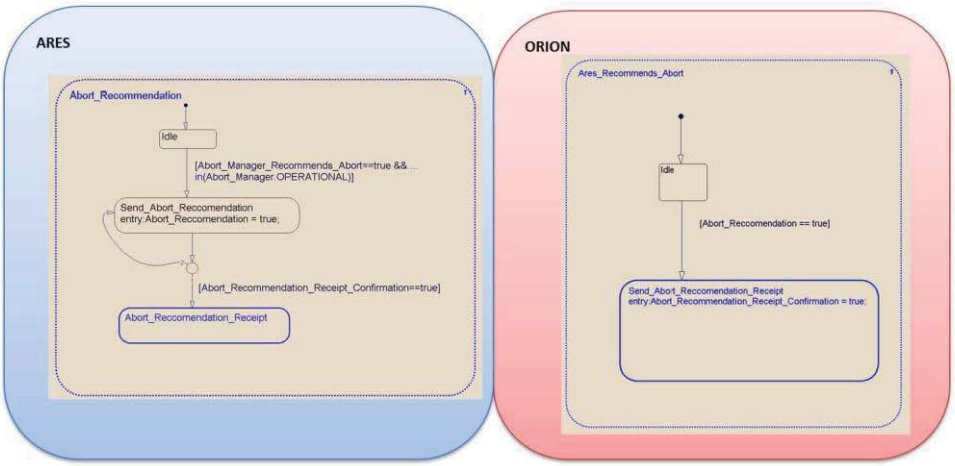


Figure A-22. Ares Sends Abort Recommendation

A-5.2.2.3 Orion Requests Auto-Safe Authority

- **Orion:** Orion requests auto-safe authority after an Ares abort recommendation is received and waits for Ares to send an Autosafe_Authority_Handoff_Receipt.
- **Ares:** Ares receives the auto-safe authority request from Orion and sends a handoff receipt and continues to monitor conditions.

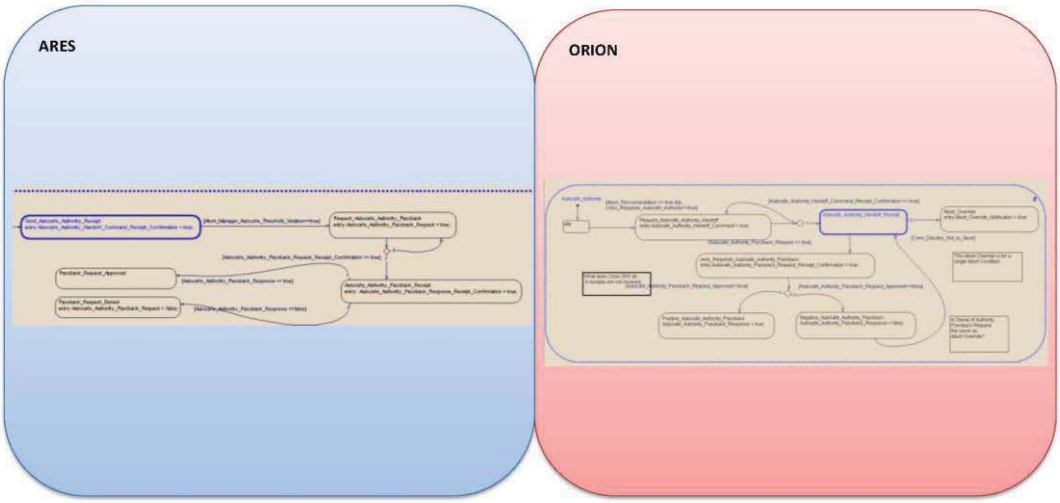


Figure A-23. Orion Requests Auto-safe Authority

A-5.2.2.4 Ares Abort Condition Exceeds Auto-safing Threshold

Figures A-24 and A-25 depict the nominal execution for abort within the model.

A-5.2.2.4.1 Orion Approves Autosafe Authority Passback Request

- **Ares:** Abort condition exceeds the autosafing threshold. Because Orion has the auto-safe authority, Ares sends an Autosafe_Authority_Passback_Request and waits for Orion to send receipt.
- **Orion:** Orion receives the Autosafe_Authority_Passback_Request and decides whether to approve or deny it. Orion approves the passback request.
- **Ares:** Upon receipt of the positive authority passback response, Ares performs safing actions.

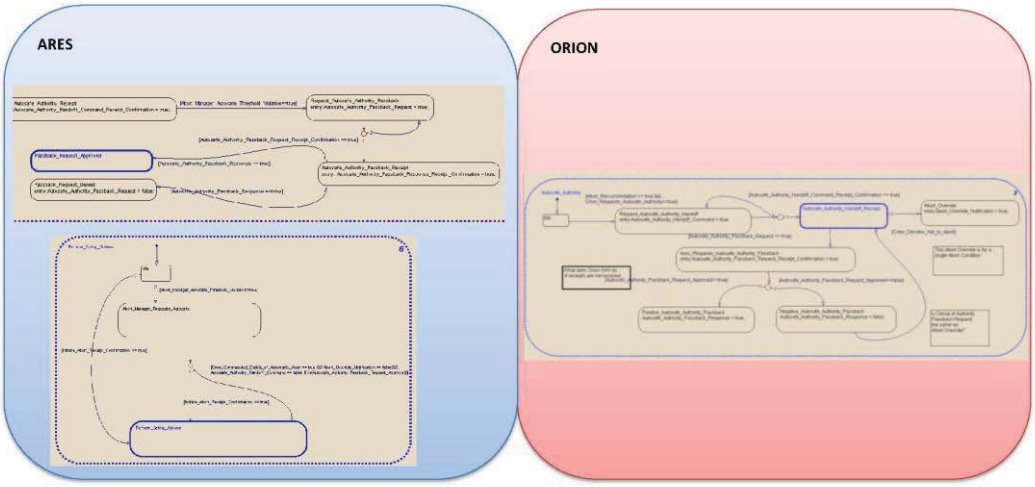


Figure A-24. Orion Approves Authority Autosafe Passback Request

A-5.2.2.4.2 Final ARES State for Nominal Abort Execution

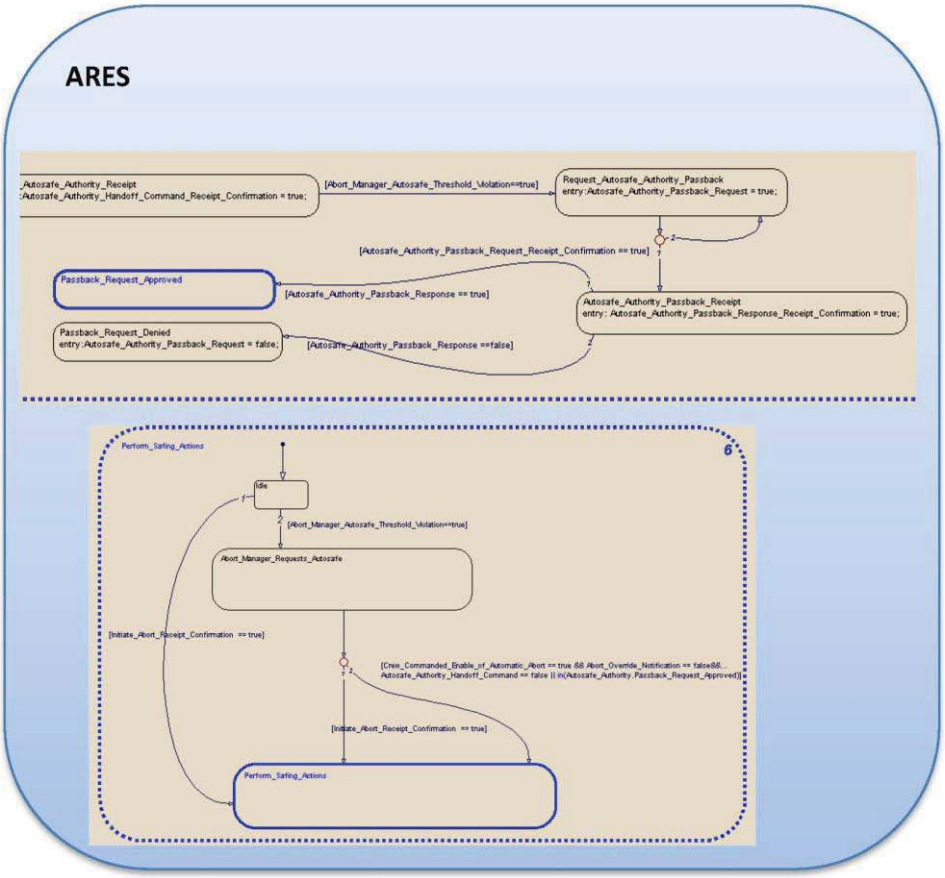


Figure A-25. Final ARES State for Nominal Abort Execution

A-5.2.2.4.3 After Orion Requests Autosafe Authority but Communication between Orion and Ares is Lost (Single Fault Scenario)

Figures A-26 and A-27 simulate a single failure loss of communication between Ares and Orion; all previous states are identical.

- **Ares:** Abort condition exceeds the autosafing threshold. Because Orion has the autosafe authority, Ares sends an Autosafe_Authority_Passback_Request and waits for Orion to send receipt (same as in nominal scenario).
- **Orion:** Because communication is severed between Ares and Orion, Orion is not aware of the autosafe authority request and remains in prior states.
- **Ares:** Ares CANNOT perform safing actions.

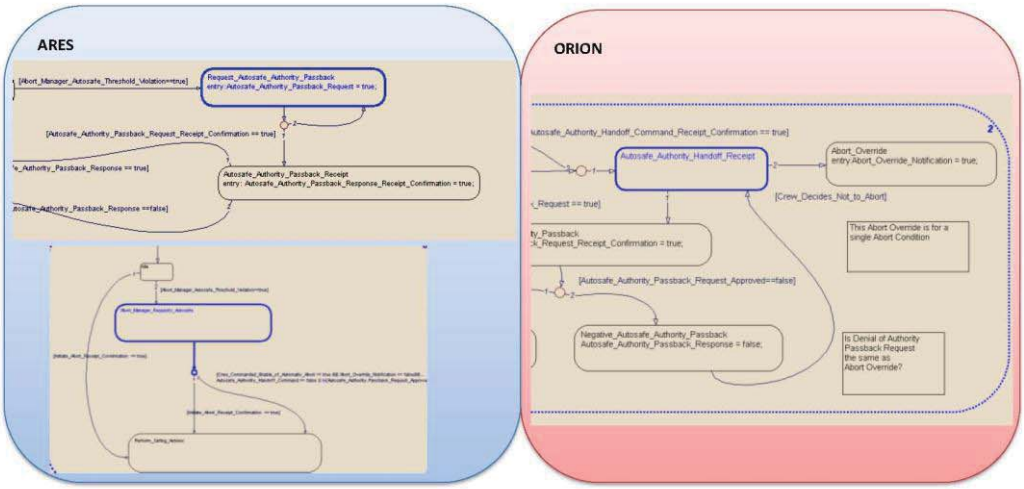


Figure A-26. Communication between Orion and Ares is Lost

A-5.2.2.4.4 Final ARES State for Single Failure Abort Execution

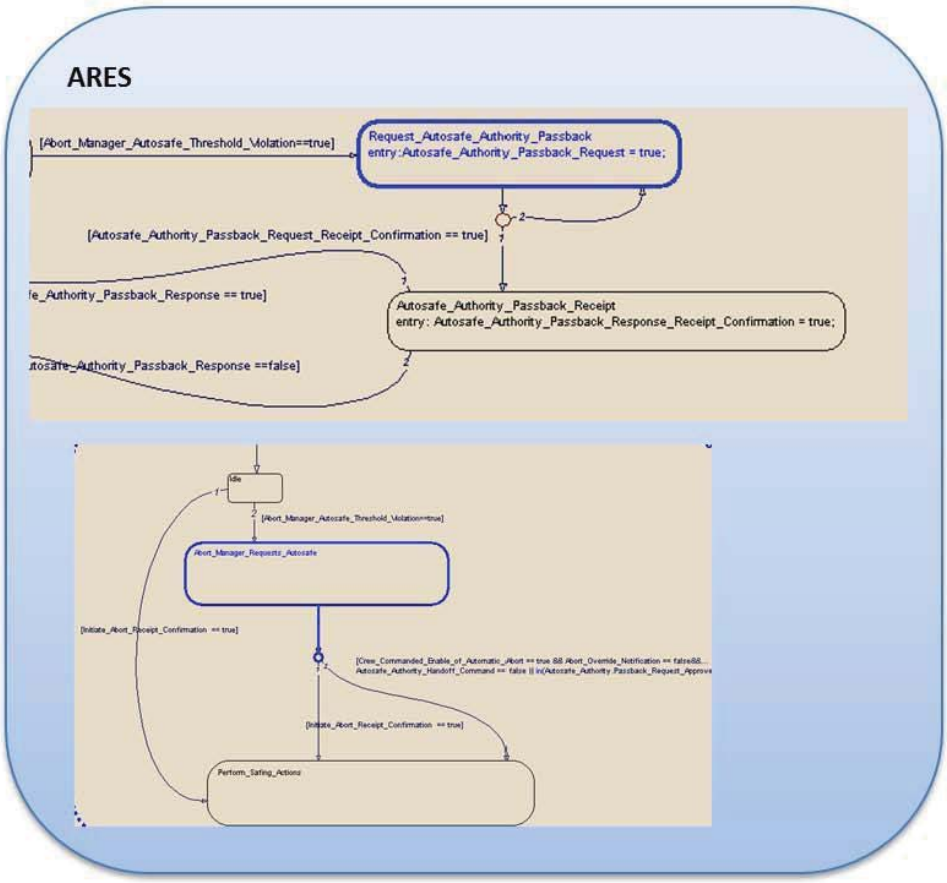


Figure A-27. Final Ares State for Single Failure Abort Execution

A-5.2.3 Verification and Validation of Model Trace

In the previous example, a possible failure to perform safing actions was determined. However, the failure could be a result of a documentation error, a modeling error, or a design error. A peer review of the model example would be required.

Note that the ability to execute the model with the model behavior animated correctly for all concurrent state machines can be more productive as a peer review tool. Without an executable model, a peer review of a static model of all concurrent state machines would require manual tracking of all states in all state machines. Executable models can help make peer reviews more productive.

Imagine trying to answer the question, "Does the abort protocol always result in an abort?" To peer review the model for all possible state sequences requires a great deal of effort and time. Model-checking tools exist that automate checking to answer this type of question. The SPIN model checker can check a model exhaustively: executing all possible state sequences and checking all paths against an assertion. If a path is discovered where the assertion is false, the example trace is reported. The assertion might be something like, "All execution paths result in an Abort." Using the SPIN model checker, several critical assertions could be run against the model to expose failure modes within the design.

A-6.0 Summation

The assessment team identified the following concerns:

No single tool chain was selected by the Ares project. No single software modeling standard was selected by the Ares project.

Teams often used tools that could not enforce consistency among different model representations. Teams often used tools that could not establish traceability among different artifacts or tools that were not fully integrated to support the entire software life cycle. Some subprojects selected certain CASE tools but failed to provide the developers with access or sufficient training in a timely manner.

Software design models included in the documents under review (e.g., ICDs, SDDs) presented models as pictures developed in PowerPoint®, Visio®, or Dia. These models were not available in an analyzable form. Not all of the tools used for modeling could support model simulation or animation (e.g., PowerPoint®, Visio®, Enterprise Architect)

Problems existed with communication, model sharing, and reuse among different teams. Inconsistencies existed between model artifacts from different tools. An SDD or ICD could contain different diagrams that were inconsistent. It was difficult to establish traceability between different artifacts (e.g., requirements and design elements).

The teams could not apply advanced V&V tools to models early in the life cycle. Requirement or design bugs could go unnoticed until the implementation and testing stages, making them more costly to fix. The selected tools made it difficult to assess if there were gaps in the overall design or the requirements. Keeping consistency and maintaining full traceability were not supported by the tools and became a manual and time-consuming process.

The full capabilities of given tools were not fully exploited (e.g., requirements management and traceability could have been managed by the CASE tool, or model validation rules could have

been defined). Lacking the capability to simulate or animate models was a source of uncertainty and ambiguity, particularly during reviews where behavioral models were inspected. It was difficult to visualize model behavior and assess the full impact of changes.

It is recommended that future Programs and projects implement a modeling plan that encompasses the following:

- **Integrated tool chain:** Use CASE tools that support model-driven software development approach as part of an integrated tool chain that supports the entire software life cycle.
- **Processes and standards:** Establish processes, metrics, and a set of standards, guidelines, and best practices for effective, model-driven software development.
- **Traceable:** Use a tool chain that supports consistency and traceability between different artifacts. Many tools provide automated support to address these issues.
- **Analyzable:** Use tools that allow the user to create and maintain models in an analyzable form. Many CASE tools out of the box provide some basic and automated model validation capability. Many analysis tools are available for the V&V of such models early in the software life cycle (e.g., model checkers, static analyzers, automated test-case generators).
- **Executable:** Reduce uncertainty and ambiguity by using executable models early on. Many CASE tools provide animation and simulation of early design models.
- **Training:** Train personnel in the effective use of the tools.

A-7.0 Acronyms

Alf	UML Action Language
ARC	Ames Research Center
ASL	Action Specification Language
CAD	Computer-aided Design
CAM	Computer-aided Manufacturing
CASE	Computer-aided Software Engineering
CSADD	CxP Software Architecture Design Document
C&DH	Command and Data Handling
COTS	Commercial off the Shelf
CxP	Constellation Program
DCU	Data Collection Unit
DEM	Data Exchange Message
ESAS	Exploration Systems Architecture Study
FDNR	Fault Detection, Notification, and Response
FGS	Fine Guidance Sensor
FSW	Flight Software
fUML	Executable UML Foundation (Foundational UML)
GN&C	Guidance, Navigation, and Control
GSFC	Goddard Space Flight Center
ICD	Interface Control Document
ISIM	Integrated Science Instrument Module
ITAR	Interational Traffic in Arms Regulations
JPF-SC	Java™ Pathfinder State Chart

JPL	Jet Propulsion Laboratory
JWST	James Webb Space Telescope
L2	Level 2
L3	Level 3
MDA	Model-driven Architecture
MOF	Meta-object Facility
NESC	NASA Engineering and Safety Center
OMG®	Object Management Group
OOA	Object-oriented Analysis
OOD	Object-oriented Design
PDCU	Power Distribution Control Unit
QUDV	Quantities, Units, Dimensions, Values
QVT	Query/View/Transformation
RSE	Reliable Software Engineering
SAMPEX	Solar, Anomalous, and Magnetospheric Particle Explorer
SAVIO	Software and Avionics Integration Office
SDD	Software Design Document
SI	Science Instrument
SIDU	Science Instrument Application Development Unit
SMSC	Specification Management Subcommittee
SoDA	Software Documentation Automation
SPICE	Simulation Program with Integrated Circuit Emphasis
SysML	Systems Modeling Language
UML	Unified Markup Language
V&V	Validation and Verification
VSM	Vehicle System Manager
XMI	XML Metadata Interchange
xUML	Executable Unified Markup Language

A-8.0 References

1. "Constellation Program Computing Systems Architecture Description Document," CxP 70078 Rev. B, August 12, 2010. [ITAR controlled, not available in public domain.]
2. "Constellation Program Orion to Ares 1 Software Interface Control Document," CxP 70091, July 7, 2009. [ITAR controlled, not available in public domain.]
3. "Constellation Program Orion to Mission Systems Software Interface Control Document," CxP 70180, September 2009. [ITAR controlled, not available in public domain.]
4. "Constellation Program Mission Systems to Ground Systems Software Interface Control Document," CxP 70187, November 30, 2009. [Sensitive but Unclassified information.]
5. "Constellation Program Orion Vehicle to Ground Systems Interface Control Document," CxP 70189, July 7, 2009. [ITAR controlled, not available in public domain.]
6. "Constellation Program Ares 1 to Ground Systems Software Interface Control Document," CxP 70190. [Sensitive but Unclassified information.]

Appendix B. Phase 2 Report

Phase II: NASA Engineering and Safety Center (NESC)
Model-Based Software Interface Management

TI-10-00609 (Phase 2)

Report Approval and Revision History

Status	Version	Change No.	Description of Revision	Effective Date
DRAFT	1.0	none	Rough draft release for internal team review	07/18/2012
DRAFT	2.0	none	Second draft release for internal team review Add model applications Add insights from Leveson, *Safeware*	09/07/2012
DRAFT	3.0	none	Third draft release for internal team review Add detail in Preliminary Design and later phases Move interface content to Attachment A Add Attachment B – Checklist of Software Interface Management Activity by Life Cycle Phase	09/26/2012
DRAFT	4.0	none	Fourth draft release for internal team review Complete summary, section B-5.0 Detail every 'ref. N'	10/25/2012
DRAFT	5.0	none	Fifth draft release for internal team review Detail model applications, various sections Detail topic sentences, various sections Additional developments needed, section 5 Detail relationships to NPR7120.5E, NPR7123.1A, NPR7150.2A, sections B-1.5 - B-1.5.1 Modeling approach evaluation, section B-3.0	12/05/2012
DRAFT	6.0	none	Final draft release for internal team review Incorporate and detail SoS integration, foundational capabilities, how can model centrism help, benefits of model-based approach, IV&V, from 2012-03 presentation	12/14/2012
FINAL	7.0	none	Final release for external review Incorporate internal team review comments Add new section B-2.0 State of the Practice	03/29/2013

Table of Contents

Technical Assessment Report

List of Figures

List of Tables

Executive Summary

NASA faces a significant challenge for integration of software interfaces within its top-level Programs. Because it is easy to create complex interfaces with software, software interfaces tend to become complex and are often called upon to handle complex behavioral and informational problems in order to provide system features, control hazards, or provide flexibility.

A successful strategy for software interface management must recognize key characteristics of software: that software exists in a conceptual realm where interfaces are not necessarily bound by physical constraints, that software interactions can occur in multiple conceptual spaces contemporaneously, and that good software interfaces do not always align to physical equipment boundaries. Additionally, the strategy must recognize that the kind of information needed for software interface engineering focuses heavily on: aggregated functions to be performed; information exchanges; coordination of behaviors; and management of system states, conditions, and performance.

Software interfaces often possess, and the engineering of software interfaces must cope with, a much higher degree and reach of interaction complexity than can be handled within human capacities for reasonable cost. For this reason, machine aids are often needed to carry out software interface engineering activities. However, there are limits to how much complexity can be managed safely even given machine assistance.

The NASA Program and project life cycle provides an overall framework for system development but is relatively unspecific as to what should be accomplished in the management of software interfaces. Each program or project must make significant choices as to where, how, and when software interfaces are addressed, choices that can in some cases be closely coupled to the project's ability to meet its goals. This report fills a gap between the high-level institutional requirements and the execution of practical actions to meet those requirements, by providing specific recommendations as to what should be accomplished at various points during the NASA Program or project life cycle.

The actions recommended fit into an overall strategy based on best practices reported in the literature. To be most effective, certain aspects of the strategy should be enacted during the earliest stages of a Program in order to be effective; even though specific software interfaces may not be identified until later design stages, the seeds for their success are sown upstream. The NASA engineering standards already recognize the importance of upstream preparation for successful software development in a general way, amplifying that recognition by recommending specific actions in each of the early stages.

The strategy recommended begins with an up-front examination during the concept study phase of the critical questions that must be answered to create an adequate foundation for software interfaces. The answers to these questions include defining the conceptual spaces within which external software interfaces will exist. Definition of the system problem to be solved at the highest level is also critical, with special attention to obtaining the kind of information needed downstream for software interface work.

The strategy then proceeds to realize desired system-level characteristics through concept development and preliminary design, with careful consideration of system physical and functional partitioning. The strategy identifies potential pathologies of software interfaces and makes some suggestions about how to correct them in these stages. The strategy continues with

elaboration of interface behaviors to a degree that greatly reduces the chances of late discovery of software incompatibilities downstream. Consideration is given to situations where software interfaces are different from the physical boundaries.

Later stages ensure discipline to, and verification of, software interfaces using the constructs developed in the early stages. System training, operation, and reuse are enhanced downstream using the software interface products developed along the way.

This paper identifies useful methods and processes and some limitations of the current state of the practice in project management, systems engineering, and model-based software interface engineering. In each of these areas, recommendations are offered regarding needed developments. Additionally, observations are provided about tool characteristics needed to support the software interface engineering effort.

Team List

Name	Discipline	Organization
Core Team		
Michael Aguilar	NASA Technical Fellow for Software	GSFC
Ronald Morillo	Senior Software Systems Engineer	JPL
Julian Breidenthal	Senior Systems Engineer	JPL

B-1.0 Introduction

B-1.1 Motivation

NASA faces a significant challenge for integration within its top-level Programs. The challenge is this: functions have migrated out of hardware into software over the last few decades, along with the attendant complexity of interaction and demands for performance. This migration has proceeded to the point where software now has vast complexity, and there is an enormous integration job to be done in the software realm. The standing situation at the Program level is that software can be the most difficult and riskiest part of the system to integrate. Some even argue that software has become the system, even for systems that superficially appear hardware intensive, in the sense that most of the desired system characteristics, engineering effort, and risk are related to software [ref. 1].

However, NASA's management and engineering requirements do not yet completely specify the work needed for software interface engineering. Software interface engineering requires particular attention, sometimes much earlier in the Program or project life cycle than might be anticipated based on historical expectations of a close coupling between hardware and software. This is especially true because satisfaction of high-level expectations of the system are usually critically dependent on software interfaces, because there are many overlapping layers of software interaction to be managed simultaneously, and because the most successful software architectures tend to blur the traditional boundaries of physical systems.

How can NASA extend its record of success to include the integration of software at the Program level? This paper proposes a strategy, which if pursued diligently, will ensure successful integration of software interfaces. This strategy is based on the recognition of the fundamental characteristics of software, on the NASA Program or project life cycle, and on the application of recent advances in the state of the practice in model-based engineering (MBE). There are

aspects of the strategy that should be enacted during the earliest stages of a Program in order to be effective; the seeds for success or failure of software interfaces are sown well upstream of the moment in the design process when specific hardware and software interfaces are identified.

To understand the proposed strategy of model-based software interface management, it is first necessary to understand the nature of interfaces and the fundamental characteristics of software interfaces. These are described below, along with the relevant background of MBE, which is believed to be essential to managing the true nature of software interfaces. The characteristics of how software interfaces affect every phase of the NASA Program or project life cycle and the specific actions that should be taken in each phase to ensure success are described. Some observations about methods and processes, about supporting tool characteristics, and about management and engineering practices that are important to the success of software interfaces are presented. Additionally, some limitations of the current state of the practice in model-based software interface engineering are presented, and recommendations are made as to needed developments.

B-1.2 Interfaces

An interface is a place where things meet and act on or communicate with each other. As illustrated in Figure B-1-1, those things may consist of matter, in which case the interface is a region—a spatial point, line segment, surface, or volume—where material things meet and interact. An electrical socket, a control stick, or the boundary between water and air are material interfaces.

Alternatively, the interacting things may not consist of matter, as is the case for software. In this case the interface is a concept, something conceived in the mind, representing an interface set where nonmaterial things meet and interact in the conceptual space in which they exist. Here, an "interface set" is an abstract group of related things in that same conceptual space, separating and enabling the interaction of the interacting things. The interfaces between system design, management, and product realization; between manufacturing and quality control; or between life support and abort decisions are examples of nonmaterial interfaces.

A distinction is made here between material and nonmaterial interfaces to illuminate some crucial aspects of software interfaces. However, this is not the only possible distinction to be made. Some authors divide interfaces into categories such as physical, functional (or logical), and flow [ref. 2]. Some include software within the physical category, as a recipient of allocations from a logical architecture [ref. 3]. Additionally, when practical systems engineering is considered, there may be innumerable other distinctions at the discipline level, for example, thermal, structural, radio, optical, power, fluids, or operational. For the present purpose, contradictions and overlap notwithstanding, the description is a means to use the material or nonmaterial distinction to reveal some critical issues with software interfaces.

Regardless of the distinctions involved, interfaces are a mechanism used to simplify the solution of complex problems. This is done by imagining a division between things while assuming that there is both a boundary region where the things interact and, at the same time, an interior region of each where they do not interact. The divisions between boundary regions are convenient because they can reduce the number of thoughts that must be managed simultaneously, can promote the division of labor between individuals or organizations, and can create a foundation for goals such as accountability, testability, reuse, and the like. The same is true of the existence of interior, noninteracting regions. The boundary regions can be used to specify requirements

and promote or focus conversations between individuals on coordinating their efforts within their respective interior regions. These concepts are illustrated in Figure B-1-2.

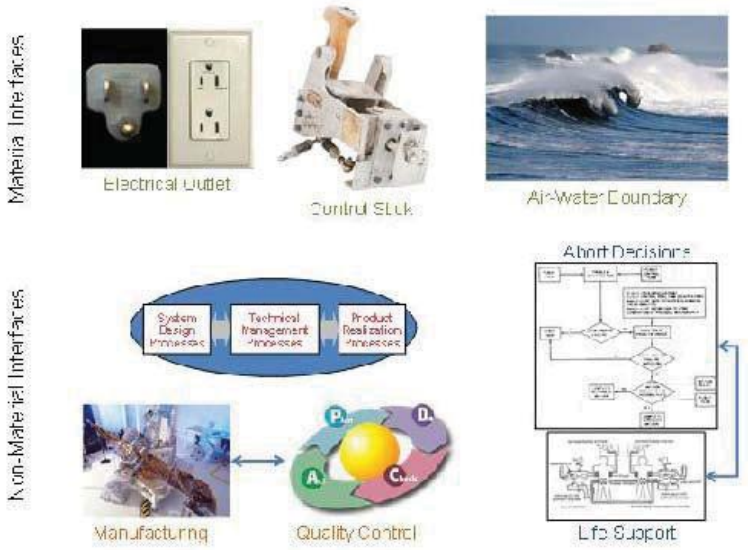


Figure B-1-1. Examples of Some Major Kinds of Interfaces

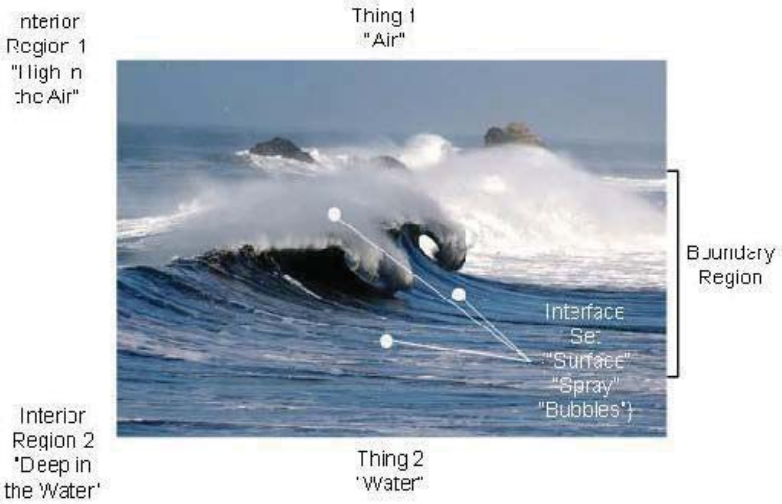


Figure B-1-2. Interface Concepts

B-1.3 Software Interfaces

Software at the very least consists of sets of instructions that operate upon certain material things: physical states of a computer, data manifested in a physical form, sensors, or actuators. It

may also be useful to broaden the concept to include other things such as data, algorithms, processes, languages, rules, decision weights, and descriptive or prescriptive supporting information. Instructions themselves are not material; neither are any of the items that might be included in the broader concept. For the purposes of this discussion, software is not material. It may have material reflections in the form of ink on paper, bits in a physical processor or memory, or pixels in an image, but this does not negate the view that software has its primary existence as a nonmaterial thing.

If software is not matter, then software interfaces are not material interfaces. Then, following from the key idea of interfaces in general, a software interface is an interface set where nonmaterial things meet and interact. Software interfaces are a mechanism used to simplify the solution of complex problems, by imagining a division between nonmaterial things while assuming that there is both a boundary region where the things interact and an interior region of each where they do not interact.

Since they are not material, it is not superficially obvious which conceptual space contains a particular software interface. A great many possibilities exist, and compared with interfaces in the obvious material world a great deal more effort can sometimes be needed to define and communicate the conceptual spaces in which software interfaces exist.

Also, there is nothing inherent to software interfaces that make them necessarily congruent with material interfaces. Indeed, some types of software interfaces exist at physical locations other than the obvious physical boundary of a piece of equipment, some types have no material interface anywhere, while other types appear at many material interfaces simultaneously. Examples of these situations are device drivers, pure software-to-software interfaces, and Web applications, respectively.

The conceptual spaces in which software interfaces exist do not necessarily conform to basic concepts from the material world such as "contact" or "partitioning." Software interfaces often exist in multiple conceptual spaces simultaneously, each of which might have its own interaction and composition rules. A good example is the use of a common variable, software component, or service; when such is used, it is possible for an individual thing (the variable, component, or service) to belong to many pieces of software. Another example is the use of objects or agents that interact in a conceptual plane above any particular piece of software; here, the agents cannot necessarily be decomposed into smaller parts—they rely on the existence of some underlying complete infrastructure that enables them to interact. Yet another example is the shared control or use by software of a single material resource.

Any of the foregoing situations can create an illusion of separation in one space for things that are in reality joined in another, tempting a naive observer to believe falsely that if there is an interface, then the things are separated and may be engineered independently. Indeed, the key assumption of boundary and internal regions for an interface may be valid in one conceptual space but simultaneously invalid in another, violating the very concept of "interface." Leveson, after Leplat, points out in her discussion of dysfunctional interactions that a disproportionately large fraction of system failures result from zones of overlap [ref. 4]. Accordingly, illusions of separation, if not managed carefully, will produce a disproportionately large fraction of software system failures.

The foundation of common thinking and language in the material world can make discussion of software interfaces difficult, error prone, and confusing. This issue was encountered in the

preceding paragraphs already, where the term "region" can be readily understood in a material context but is formally meaningless in conceptual spaces that do not conform to the partitioning rules of the material world. Completely correct reasoning necessitates the awkwardness of a term like "interface set," but the awkwardness is severe enough that most people will revert to the term "region" to maintain an adequate level of communication of other more important concepts. It does not always create an error, but there is always a risk of either dragging concepts out of the material world into conceptual spaces where they do not apply or of failing to recognize that the software interface needs to be expressed using nonmaterial concepts.

Leveson has argued that interfaces in hardware tend to be simpler than in software because physical constraints discourage complexity [ref. 4]. Costs associated with hardware complexity are immediate and obvious because increasing the complexity of physical interfaces greatly increases the difficulty of design and construction. Also, the physical separation between physical components tends to reduce the degree to which they interact and simplify interactions when they do occur. On the other hand, with software it is easy to make anything depend on anything else. Adding complexity to software interfaces is easy in the short run because these interfaces are subject to minimal physical or cost constraints, and those cost consequences that do exist are delayed. Thus, there is a relatively flat route to complexity in software interfaces, which increases the chance that they will become complex.

There is also the issue of management and engineering processes for software interfaces: these processes can attenuate or amplify the real difficulties inherent in software interfaces, depending on the extent to which such processes are rooted in the assumed properties of the material world or the extent to which they recognize, tolerate, and promote the necessity of managing and engineering a nonmaterial world.

As will be discussed further below, the distinguishing characteristics of software interfaces—nonmateriality; conceptual spaces; potential for deviation from physical interfaces both in location and inherent properties; the potential illusion of separation; difficulties of communication; ease of creating complex interfaces; and availability of suitable processes—have profound ramifications for software interfaces. These characteristics form a foundation for success or failure of software interface work and for the recommendations in this report.

B-1.4 MBE and Model-Based Systems Engineering (MBSE)

There are a great many definitions in use of what is a model, but for the purposes of this report a model is any incomplete representation of reality. One way of categorizing models is to say that they may be physical, quantitative, qualitative, or mental, and that they may be applied for definitive, descriptive, or normative purposes [ref. 5]. Another way is to categorize them as physical or abstract, with the abstract type further divided into geometric, logical, and quantitative, which themselves may be related to discipline or domain-specific models [ref. 17, slide 5]. In the case of MBE or MBSE, it is usually the quantitative or qualitative or abstract types of models that are meant.

The National Defense Industry Association (NDIA) [ref. 30] defines MBE as an approach to engineering that uses models as an integral part of the technical baseline that includes the requirements, analysis, design, implementation, and verification of a capability, system, and/or product throughout the acquisition life cycle. The International Council on Systems Engineering (INCOSE) [ref. 40] defines MBSE as the formalized application of modeling to support system requirements, design, analysis, verification, and validation activities beginning in the conceptual

design phase and continuing throughout development and later life cycle phases. INCOSE views MBSE as part of a long-term trend toward model-based approaches adopted by other engineering disciplines, including mechanical, electrical, and software. A "system model" usually consists of a functional or behavioral model, a performance model, and a structural or component model [ref. 17, slide 16]. A system model has specification and integration relationships with other types of discipline-specific models, such as software, hardware, cost, safety, thermal, communication, loads, and the like [ref. 3, figure 2.2].

Mental models are always used in engineering. In non-MBE, the mental models may be exchanged and compared between engineers orally, in writing, or simply kept hidden with only end results presented. A key difference in MBE is that models are also expressed explicitly in a machine-usable form outside the engineer.

The use of machine-usable models in engineering opens up several major possibilities that are not accessible in non-MBE, specifically that machine aids can be used to:

- Check models for certain types of completeness, consistency, and accuracy
- Exchange information between models and modelers
- Maintain consistency between models, both by identifying and correcting inconsistencies and by efficiently propagating changes
- Derive system characteristics, such as performance, behavior, resource usage, or cost

The ability to use machines for these functions greatly enhances the speed, accuracy, range of knowable and controllable characteristics, and level of complexity that can be achieved. As the expression of models for machine use entails the use of extremely precise languages—much more precise than natural language—semantics can be clearer, and the issues of ambiguity, error, or confusion in communication are much more easily resolved.

There are additional benefits that accrue as a result of a model-based approach. For example:

- There can be greater consistency of all products because any single piece of design information can be expressed authoritatively in a single place that can later be referred to by others for decisions, derivations, or formation of artifacts.
- There can be greater congruence between documentation and reality:
  - Model-based artifacts can be generated automatically, lowering the cost to keep them up to date because information does not have to be hand copied from one artifact to another, with the result that artifacts can always match the best available information.
  - There can be less divergence between intent of designers and what is built, owing to machine-assisted product realization during late stages of the product life cycle.
  - Models can give greater visibility into what the software is really doing.
    - "Only the computer knows for sure, and now it's talking to us."
- Access bottlenecks can be reduced primarily because traceability of information can be better in the model-based approach, but also as a side effect of the manner in which model-based techniques produce greater congruence between reality and the information available, and because many people can have access to the information they are authorized to have, more quickly and on an as-needed basis, without going through manual distribution or search processes.
- Models can give an enhanced ability to detect large blunders early (if applied diligently from the beginning) because of an increased ability to determine the consequences of decisions with automated analysis or simulation.

- Models used for verification can have higher quality and provide greater confidence if design and manufacturing models are applied diligently before and after use of the verification models.
- There can be better execution of appropriate practices because models can make deficiencies of practices visible sooner.
- Models themselves can assist in revealing hidden flaws of the models.
- There can be lower lost investment in erroneous design because the model can reveal flaws earlier in the process, enabling correction before downstream work is done—work that would be invalid if the upstream mistake were not corrected immediately.
- Faster, more accurate results may be possible even under downward labor and schedule pressure.
- Lower costs for verification because of fewer inconsistencies between artifacts.
- There can be lower investment in preparation of verification tools, because model inputs are already available when needed.

These general advantages are significant for software interface work. It is fairly common in software work that there is too little time, there are too many gaps or inconsistencies, the complexity that can be managed is too low, the propagation of change is too difficult to manage, or the expected characteristics of the system are too difficult to derive or control at the needed level of accuracy. When faced with the problem of defining and communicating the conceptual spaces within which software interfaces exist, natural language often fails; the problem is even greater when relationships between interfaces in different spaces must be engineered. The use of MBE can overcome these challenges to a greater degree than manual methods.

The state of the art in both MBE and MBSE is rapidly evolving. Models are frequently being used in most disciplines. However, according to a recent NDIA report [ref. 30] there is currently poor integration of models across the life cycle and limited reuse of models between programs. There is a variation in modeling maturity and integration across engineering disciplines (e.g., systems, software, mechanical, electrical, test, maintainability, safety, security). Mechanical and electrical computer-assisted design (CAD) and computer-aided engineering (CAE) are fairly mature with industrial, academic, and standards bodies, but the systems, software, and test disciplines are fairly immature. There are many MBE activities across industry, academia, and standards bodies. The modeling standards are evolving (e.g., core manufacturing simulation data, and modeling languages such as the Systems Modeling Language (SysML), United Profile for DoDAF and MODAF (UPDM), Modelica, Architecture Analysis and Design Language (AADL), and tools are evolving toward an MBE paradigm and progressing toward greater tool-to-tool interoperability.

A recent report by Friedenthal [ref. 17, slide 19] notes that current basic MBSE practices include tracing requirements to the system model; behavior representation with use cases and scenarios; structure modeling with architecture block diagrams; analysis models for mass, power, and cost rollups; verification models with test cases; and management models including process models, point-to-point tool integration, document generation with simple document templates, and configuration management based on check-in or check-out capabilities. Advanced practices currently include requirements management support for structured text, behavior representation with executable models and fault models, structure models with detailed interface models and variant modeling, integrated analysis tools, model-based verification planning and test architectures, model-based management metrics and plans, model or tool integration with model

transformations and XML (extensible markup language) model interchange (XMI), rapid customization of document generation, and model-based product life cycle management integration.

B-1.5 The Software Interface Life Cycle

B-1.5.1 The NASA Program or Project Life Cycle

The NASA Office of the Chief Engineer (OCE) [ref. 21] defines the major NASA life-cycle phases as Formulation and Implementation. For flight systems and ground support (FS&GS) projects, the phases are further divided into the following seven incremental pieces, widely referred to as phases:

Formulation

Pre-Phase A: Concept Studies. Devise various feasible concepts from which new projects (programs) can be selected.

Phase A: Concept and Technology Development. Fully develop a baseline mission concept and begin or assume responsibility for the development of needed technologies.

Phase B: Preliminary Design and Technology Completion. Establish a preliminary design and develop necessary technology, including a complete set of system and subsystem design specifications, sufficient to establish a firm cost and schedule.

Implementation

Phase C: Final Design and Fabrication. Establish a complete design, fabricate or produce hardware, and code software in preparation for integration.

Phase D: System Assembly, Integration and Test, Launch. Assemble, integrate, and verify the system, prepare for operations, and launch.

Phase E: Operations and Sustainment. Conduct the prime mission, meet the initially identified need, and maintain support for that need.

Phase F: Closeout. Dispose of systems, analyze returned data and samples, document lessons learned, and archive data.

The incremental phases are separated by "key decision points (KDP)", events at which the decision authority for a Program or project determines the readiness to progress to the next phase. The phases are summarized for a project in Figure B-1-3; there are variations for Program life cycles depending on the type of Program, which are described more fully in reference 22.

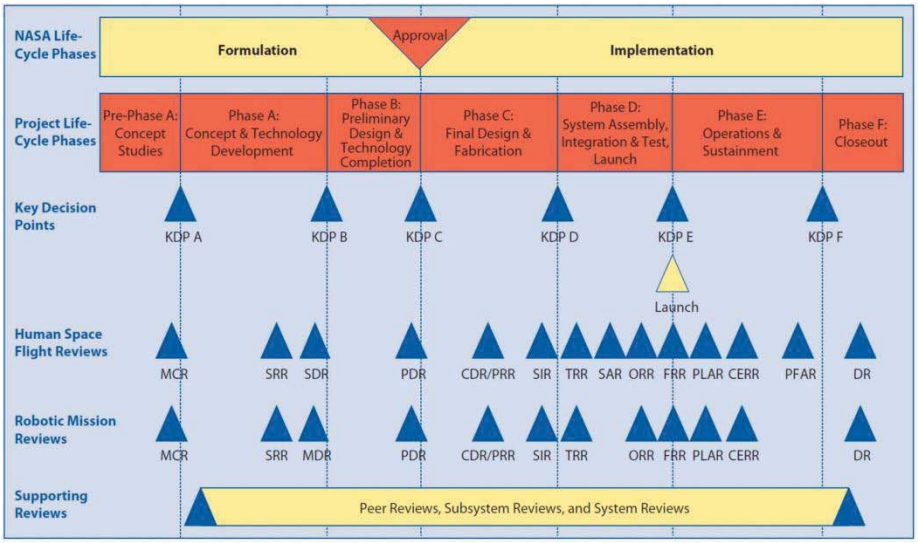


Figure B-1-3. NASA Project Life Cycle

The NASA OCE also adopts the view that project management can be thought of as having two major areas of emphasis, both of equal weight and importance: systems engineering and project control [ref. 22, section 2.0]. Some of the activities of these two areas belong to both, including planning, risk management, configuration management, data management, assessment, and decision analysis. The system design, product realization, and other technical management activities of project management can be thought of as part of systems engineering.

B-1.5.2 The NASA Systems Engineering Engine

The NASA Systems Engineering Processes and Requirements [ref. 23] describes systems engineering using the concept of a "systems engineering engine." The engine carries out 17 common technical processes organized into three sets: system design, product realization, and technical management. The interactions and flows within the systems engineering engine are illustrated by Figure B-1-4.

The processes within the systems engineering engine are used both iteratively and recursively. They are used iteratively in the sense of repeatedly applying processes to the same set of products to either correct a discovered discrepancy or to incrementally mature the system definition to satisfy progress goals for successive life-cycle phases. They are used recursively in the sense of repeatedly applying systems engineering processes to each product of a system hierarchy from the top to the bottom, until the lowest products are defined to the point where they can be built, bought, or reused. Recursion is also used to integrate the smallest products into larger products until the whole of the system has been completed.

Thus, the systems engineering engine is used to execute much of the project management assignment, in a way that involves a successive refinement of the solution over the life-cycle phases, and in hierarchical detail. Even during the early phases, phase-appropriate products are developed at lower levels and then integrated up to the top tier of products for transition to the

customer. For software interfaces, the phase-appropriate products begin with interface expectations and external constraints, and then mature into interface requirements and descriptions, into experimental and test interfaces, and finally into actual operational interfaces. Every phase can be supported by models of the interfaces, with increasing fidelity at each phase.

B-1.5.3 Software Interfaces in the NASA Project Life Cycle

Software interfaces do not appear explicitly in the NASA project life cycle. However, software is mentioned many times in the project management requirements. In general, the NASA standards consider software to be one possible component of a system, and express expectations that software development will be fully integrated with the general systems engineering effort.

It may be surprising that the NASA standards require attention to software beginning from the conceptual study phase. For example, the project management requirements [ref. 21] expect the formulation agreement to state the prototypes and software models to be built and to identify software assumptions. The Program or project plan is expected to identify leading indicators for software. The safety and mission assurance plan is expected to clarify how software is handled and to explain approaches to software problem reporting, analysis, and corrective action. The acquisition plan is expected to identify major acquisitions, including software development. The Mission Operations Plan is expected to describe how the Program will implement software. All of the requirements require preliminary awareness of major software components by the time of the System Requirements Review (SRR). The mission description is required to describe the software to be developed. Preliminary software configuration management is required by SRR. Taken as a whole, these requirements require a preliminary awareness of software by the Mission Concept Review (MCR) and a substantial awareness—to the point of knowing what software needs to be acquired and managed for configuration—by the SRR.

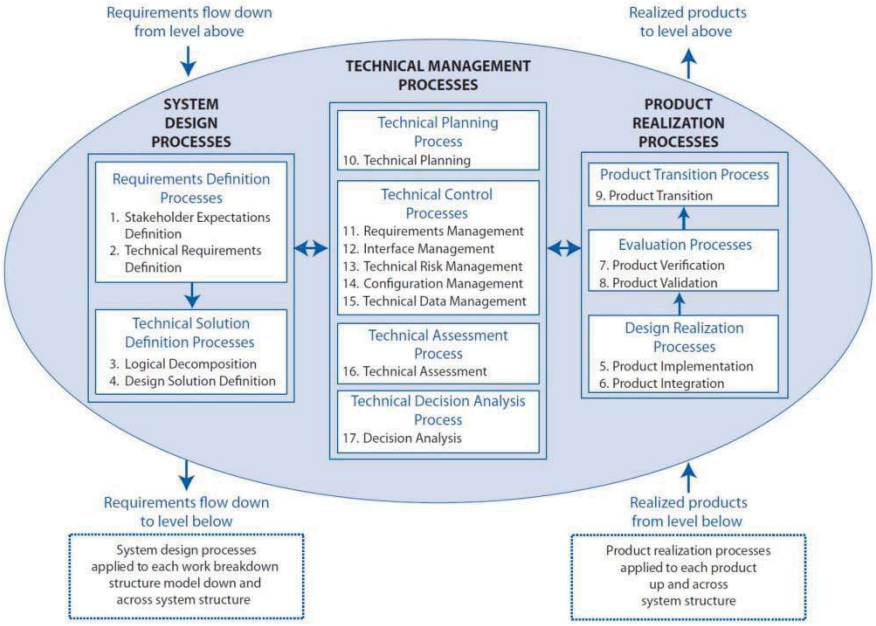


Figure B-1-4. NASA Systems Engineering Engine

The Program or project management requirements include interfaces in general as part of a project's architectural control document and also under the heading of Program architectures. The formal definition of interface control documents (ICDs) in the requirements emphasizes physical interfaces; if there is a software ICD, it must be inferred to be part of the type of interface called "any other interface." Interfaces down to the subsystem level are to be identified in the formulation agreement, and the Program or project plan is required to identify system interfaces. Taken as a whole, these requirements require an awareness of software interfaces at the system and subsystem level beginning at MCR, with the formulation agreement baseline.

The NASA Software Engineering Requirements, NPR 7150.2A [ref. 24], gives requirements on representation of software interfaces in more detail, although it does not specify exactly when the requirements must be met. If the project has safety-critical software, it is required to identify safety-critical interfaces. The requirements recognize the activity of defining interfaces as part of software design; typical views captured in an architectural design include definition of external and internal interfaces. Interface design description is required, and its content is specified in Chapter 5 of reference 24. Software requirements specifications are required to specify interface requirements for each computer software configuration item (CSCI); combined with the requirement in reference 21 to identify subsystem interfaces in the formulation agreement and combined with the requirement to identify high-level software components as part of acquisition planning, this may imply preliminary interface specifications for the highest level CSCIs as early as the MCR or SRR, and certainly by the time acquisition of software development gets under way. Interface designs are required in the software design description; combined with the requirement in reference 21 for architecture descriptions in the Program

commitment agreement and in the formulation agreement and the requirements for architectures at MCR, SRR, System Design Review (SDR), Mission Definition Review (MDR), and later reviews with increasing levels of maturity, it is clear that software architecture and interfaces should be appearing at the earliest life-cycle stages.

The NASA Systems Engineering standard [ref. 23, section 5.2.1.6] requires the technical team to ensure that system aspects represented or implemented in software are included in all technical reviews, to demonstrate that project technical goals and progress are being achieved, and that all NPR 7150.2 software review requirements are implemented. For the types of systems to which this report applies, this requirement implies including software from the earliest reviews. The systems engineering standard [ref. 23, section 6.2.6] also requires software development to be fully integrated with the technical effort. The definition of a system [ref. 23, section A.26] affirms software as being seen as part of the system. The systems engineering standard [ref. 23, section C.1.c] expresses an expectation that systems engineering processes will comply with the software product realization requirements in reference 24. The systems engineering standard [ref. 23, section C.1.4.4] also expresses an expectation that the design process will partition requirements to software; combined with the notion of executing the design process before the MCR expressed in reference 22, section 2, this implies partitioning requirements to software beginning at the MCR. Product validation includes an expectation that software should be included in validation, which in turn would begin by MCR. The entrance criteria for the PDR or SDR include description of software to be developed. The entrance criteria for the SRR, MDR, SDR and later include requirements for a system software functionality description. Taken as a whole, these requirements require early treatment of software interfaces.

B-1.5.4 Origin of Software Interfaces in the System Problem and Boundary

In some systems, software is the system. This can be true if the system is a software system, obviously. It can also be approximately true even when the hardware is immense, if the software has substantially more complexity than the hardware. The interactions between the physical elements of a mission control center and a launch system, a launch vehicle and a payload, or one vehicle and another may exist, but represent such a small fraction of the total body of interactions involved in the associated software as to be a minor factor in the total systems engineering effort. Exceptions to this situation would be systems that do not have software, that only have software that possesses little interaction, or that are primarily composed of people, methods, and policies. However, software-intensive systems are the scope of this report.

If the system is one where substantial parts of the system problem are to be solved by software, then the complexity represented in the system problem will eventually be contained within the software regardless of which physical element ends up containing the software. Therefore, definition of the software begins with defining the system-level problems to be solved. Correspondingly, definition of the software interfaces begins with defining the system boundary.

B-1.5.4.1 Realization of system-level "emergent" characteristics through software

Solution of the problems to be solved entails the system at the top level, realizing some new characteristics that emerge from relationships between the sub-elements. These should be characteristics that would not otherwise occur from the mere juxtaposition of the sub-elements.

If the emergent characteristics are to be engineered, that is, to be contrived through skill and craft rather than occurring as a mere happenstance, then the engineer must know both the desired and

undesired emergent characteristics and then contrive the sub-elements of the system to provide those characteristics.

If most of the problem to be solved by the system is going to be solved by software, then most of the contriving—the engineering—will be applied to the software. If this proportion is neglected, then the software effort stands a poor chance of solving the intended problems compared with its hardware cousin. The same may be said of the software interfaces.

B-1.5.4.2 Development through system architecture

Software interfaces within the system arise from choices made in partitioning the system architecture. Any partitions created by the architecting process create interfaces, and these will be identifiable as software interfaces in one of two situations: the partitions are software, or the partitions are components that contain software. The functions that solve the top-level system problem will be allocated to the partitions, and if the partitions are or contain software, then some aspects of the functions will be allocated to software interfaces. These interface requirements will be closely tied to the inputs and outputs of whatever functional partitions arise in the architecting process.

B-1.5.4.3 Dependence on system partitioning and allocation process

In the previous section, it was stated that software interfaces *may* receive functional allocations in the architecting process. However, it is also true that software and software interfaces must receive functional allocations. Otherwise it is not possible to know what to expect of the software, what to require, or ultimately to know whether any software produced is acceptable.

The choice of partitions can have a profound impact on the achievability of the interfaces that result. Various pathologies can arise simply because the partitions are placed in the wrong place. For example, placing a boundary around a modem that requires the user to know the precise communication settings for communication to occur can create extra interfaces just to communicate that information. Or, placing a boundary around a control system that requires the exchange of large quantities of high-rate and precise actuator information can create an interface that is extremely complicated and expensive to implement.

B-1.5.4.4 Dependence on software system design

The software interfaces depend on the effectiveness of the software system design at solving both the high-level problems to be solved by the system, and the allocated subproblems within the system that contribute to solving the high-level problems.

B-1.5.4.5 Dependence on capabilities of engineering environment

The engineering environment has to be able to support software engineering, not just hardware engineering.

The engineering environment includes people (managers, engineers, acquirers, testers, operators, maintainers, lawyers, financers, trainers, etc.), hardware used for engineering, software used for engineering, policies for engineering, methods of engineering, and information involved in engineering. If this environment does not support software engineering, then software engineering cannot be done well.

If software engineering cannot be supported by the engineering environment, then software interfaces cannot be done well.

B-1.6 Summary of the Issues

B-1.6.1 Every Project Decides Where, When, How, and How Much Software Interfaces Are Addressed

The NASA Program or project life cycle does not specify at which tier of the system breakdown structure software interfaces will be handled. The life cycle does specify that software interfaces should be handled in early phases, though the degree of detail is an open question. The exact method of addressing software interfaces is undefined by the requirements, making this a decision that each program project has to make individually.

B-1.6.2 Software Interfaces are Driven by the System Problem and Boundary

If the system problem contains a challenge that will eventually be met by means of software, successful development of the software interfaces will depend on relating that part of the system problem to the interfaces as they are eventually created. Similarly, if the system boundary entails behavior, information, or media exchanges with the environment that can only be satisfied using software, then the software interfaces will depend on relating the system boundary to the software interfaces.

Since the software interfaces depend on the system problem and boundary, information appropriate for engineering of software interfaces needs to be obtained in the context of defining the system problem and boundary. This information is in addition to the information usually needed for hardware, although it is certainly possible that hardware information such as size, mass, or power constraints may eventually lead to software interface design constraints, depending on how the architecting process plays out. Suggestions regarding the kind of information needed for software interface engineering appear below in section B-3.1.1.

B-1.6.3 Good Software Interfaces May Not Align with Good Physical Interfaces

It is widely agreed that good interfaces should be simple, facilitate delegation of the problem to appropriate kinds and sizes of organizations, satisfy all the relevant requirements needed to solve the system problem, and match what the interfacing parties can supply or use. It is generally agreed that good interfaces should isolate internal details, be robust against failures in either the environment or the system, and facilitate system upgrades, changes, or interoperability. The challenge comes when achieving any of these goals for software interfaces conflicts with achieving them with hardware interfaces. Software clients, hardware drivers, software services, scripts, applications program interfaces, networking protocol drivers, and the widely used Hypertext Transfer Protocol (HTTP) and Hypertext Markup Language (HTML) are all examples of solutions that address such a software or hardware interface conflict by placing the software boundary at a different location than the hardware boundary.

B-1.6.4 Sometimes Conceptual Spaces for Software Interfaces Need Definition

Software interfaces by their nature occur in conceptual spaces rather than physical ones. In some cases it may not be necessary to define the conceptual spaces specifically for a particular Program or project. For example, if the conceptual spaces are widely known, as in the cases of "email," "Internet protocol," or "Web page," then they can safely be taken as a given for anyone likely to be working with the software interfaces. In other cases, it may be necessary to fully describe the conceptual space in which the software interface exists: the things that inhabit the space, the kinds of relationships that exist in the space, and the "interface sets."

B-1.6.5 Software Interfaces Can Create an Illusion of Separation

A single software interface can exist in multiple conceptual spaces simultaneously. For example, passing a datum from one application to another can exercise all layers of the Open-systems Interconnection model [ref. 41]. For another example, passing an email message from one person to another can exercise application, middleware, and operating systems, as well as the same on intervening communications systems. In both of these examples, each of the underlying layers or software systems has interfaces in the conceptual spaces peculiar to their particular system problem and boundary. All of these are operating contemporaneously and can create unanticipated coupling between things that are believed to be separated. Hence, the separation created by software interfaces can be an illusion.

B-1.6.6 Sometimes Software Interfaces Show Up Early at a High Level

Software interfaces may show up at higher levels in the system hierarchy than the level at which they are eventually implemented, and early in the development process. This may happen directly, because it is obvious from the outset that an interface with the environment is going to be a software interface. It may happen because a high-level allocation of software function is made between major physical components, entailing creation of a high-level software interface between the components. Or, it may happen because interfaces between high-level functions are identified early, and those eventually are deployed on a software system. Finally, it may happen because the system itself is a software system. In any of these cases, the NASA management and systems engineering requirements expect software interfaces to be addressed early.

B-1.6.7 Software Interfaces Are Always Part of a System Function

Every part of a system should be addressing one or more functions of the system; software interfaces are no exception. For software interface engineering to occur, it is necessary to know to which functions a software interface is contributing and, conversely, how each function involved depends on the software interface.

B-1.6.8 Software Interfaces May Need Computer-Aided Engineering

The potential for complex, multi-layered conceptual spaces for software interfaces, coupled with the typically large number of system functions in which any given software interface is participating, makes it common that the complexity of the software interfaces exceeds that which can be managed within human capability alone for reasonable cost. In these cases, it is necessary to use computer aids for software interface engineering.

B-1.6.9 Software Interface Engineering Relies on the Engineering Environment

Software interface engineering relies on the personnel, organizations, facilities, methods, processes, policies, information, and tools composing the engineering environment. If software interface engineering is to be successful, then the engineering environment must be capable of handling all of the major features of software interfaces: nonmateriality, conceptual spaces, potential for deviation from physical interfaces both in location and inherent properties, the potential illusion of separation, and the difficulties of communication.

B-1.6.10 Software Interface Complexity Won't Go Away on Its Own

The barriers to creating complex software interfaces are low; it may even be easier to create complex interfaces than simple ones. Logical connections are easy and cheap to introduce in the short run, and there is a negative tradeoff between software module complexity and software

interface complexity. That is, separating a complex piece of software into smaller modules will make the modules simpler but will increase the number of software interfaces, and these interfaces will become more complex to handle the larger number of smaller components [ref. 4].

Furthermore, there are limits to how much complexity can be handled safely, even given the use of machine aids.

Therefore, it is necessary to introduce discipline and training to avoid or mitigate complexity, to evaluate the software interface complexity associated with choices in the architecting process, and to make hard choices to simplify the system under consideration. These choices may include removing drivers on complexity by removing system features, finding alternate ways to control hazards, or reducing the required degree of flexibility.

B-2.0 State of the Practice

This section provides an overview of the current state of the practice in MBE and MBSE, providing a foundation for practical objectives during the system development life cycle that are recommended later in this report.

B-2.1 Supporting Program Integration – Design

Model-based techniques are developing rapidly in five major areas within the field of system-level design:

- Capturing architecture, design descriptions, and behaviors in models
- Program-level definition of system and software interfaces
- Program-level design analysis
- Modeling integrated performance
- Maturing model-based products and activities in a collaborative environment

Each of these areas will be discussed with respect to issues currently challenging the community, assessment of the capabilities and needs in the area, examples of the state of the practice, and recent lessons learned if available.

B-2.1.1 Capturing Architecture, Design Descriptions, and Behaviors in Models

B-2.1.1.1 Issues

Experience with the Constellation Program revealed that program-level integration suffered from the lack of timely and accurate design descriptions, especially in support of key milestone reviews. Typically, design descriptions were arriving in the final days before the review began, with thousands of pages to be reviewed in a couple of weeks and a low level of formal correspondence between views presented by the many authors. Given the high level of complexity in the software realm, true technical reviews are impossible without machine-based design checks and dynamic exploration of design. Typically, the milestone reviews ended up more often reviewing document formats rather than the technical contents.

The review deliverables are never complete; they are just a snapshot taken at a particular point in time that then go on to evolve dynamically as engineering work proceeds.

Where possible, there is a need to address life cycle phase "handoffs," for example, requirements to design, implementation to validation and verification, or delivery to operations.

There is a need for software designs to address both nominal and off-nominal behavior; for example, aborts is a huge area of focus (although there is justification for considering many other off-nominal scenarios).

There is an increased need for interoperability of systems, in particular for cooperative developments.

There is a need for type checking support for activity modeling, many views and viewpoints, and support for patterns and reuse.

There is a need to define model interchange capability to support integration.

The NASA OCE systems engineering vision includes a desired future engineering environment, which contains model based artifacts, seamless data flow, and distributed teams [ref. 48].

B-2.1.1.2 Assessment

Capturing architecture and design descriptions in system models brings a number of desired advantages:

- Single source of authoritative information; information entered only once [refs. 14, 20, 44, 48, 49]
- Clearer semantics [refs. 49, 50, 51]
- Models checked automatically for certain types of consistency and completeness issues [refs. 39, 50]
- Autogeneration of standard documents; standard documents are kept up to date [refs. 12, 15, 16, 17, 39]

There is a need to describe many views and viewpoints, all based in a single source of authoritative information.

There is also a need to be able to "stitch" together composite views [ref. 12].

There are problems with the implementation of standard exchanges in different tools. XMI is a standardized format for model interchange between Unified Markup Language (UML)-based tools, but in practice an XMI file exported from one tool is unlikely to import correctly into a different tool.

B-2.1.1.3 State of the Practice

Friedenthal [ref. 17] reports that basic practices are currently tracing requirements to models; representing behavior with use cases and scenarios; using architecture block diagrams for structure models, analysis models for mass/power/cost rollups, test cases for verification, process models for management, point-to-point tool integration, document generation from simple templates, and check in/check out for configuration management. Advanced practices are structured text for requirements, executable models and fault models for behavior, detailed interface models and variant modeling for structure, integrated analysis tools, verification planning and test architectures from models, management with metrics and generation of management plans, model transformations and XMI, rapid customization of documents, and product life cycle management integration.

Bayer *et al.* [ref. 14] describe an Architecture Framework Tool (AFT) to capture high-level design description, allowing the engineering team to capture architecture information without

investing in the SysML learning curve; the tool is compatible with SysML models through a common ontology.

Ingham *et al.* [ref. 12] report a series of model-based capability developments focusing on specifying system behaviors for critical activities. Ingham's team uses functional modeling to focus on execution of spacecraft functions using activity diagrams, state charts, and sequence diagrams; they are also working effectively on off-nominal behavior patterns and operational scenarios. The team has already developed a MagicDraw® DocGen model of their functional definition document. This enables them to use queries to get information from System Model and embed information into documents, which then allows fast compilation to a DocBook standard representation, which is easily translated to Portable Document Format (PDF) or HTML. The result is that they are able to generate the most up-to-date version of the document with a few button clicks. The System Model can have multiple DocGen document models, all of which refer to the same source material, meaning that there is no need to maintain numerous disparate sources. Future developments include investigating autogeneration of systems engineering artifacts from the system model, for example, functional descriptions, ICDs, activity plans, operations handbook, as well as different views in which "complete Initial spin-up" and "complete rate adjustment to high rate" are merged into one diagram, with the hope of automating the process of stitching together such views.

Bindschadler [ref. 49] describes an architectural evolution activity in which his team identified fundamental patterns in design, based on varying perspectives and views. From this activity, they are leveraging design patterns to automate modeling of common elements of design. In particular, they have implemented the concept of a modeled timeline, which is important to the verification of critical program activities like an integrated abort.

Moody *et al.* [ref. 44] report the development of the Orion system architecture model (SAM), an analytic model representing vehicle function, structure, and behavior. In this model, models of vehicle function are system capabilities without operational context, models of vehicle behavior are functions in their mission context, and models of vehicle structure include system and component hierarchies as well as interface/interconnect definition. At present, the SAM is not part of the system development cycle, it is documentation of the system and follows design.

Arteaga [ref. 52] is creating a SysML model for the Unmanned Aircraft System (UAS)–National Airspace (NAS) integration project. The model includes integrated mission, functional, and physical models. The mission model provides a concept of operations for nominal and off-nominal scenarios, and capability and functional models utilized in missions. The architecture elements include unmanned aerial vehicle, ground control system, and air traffic control traffic (with all mentioned system aspects). They have developed UAS-specific ontologies. The main characteristics of their modeling effort are a scalable model structure and organization that includes model annotations and external references, as well as various examples of telemetry packets and data flows to model interfaces, abstraction levels, and functional, system, and operation views.

Crain [ref. 44] has developed a tool-independent MBSE approach based on NASA Standards 7123, 6105, and 7120 out of initial work on the Exploration Program models, which supports the end-to-end NASA life cycle from stakeholder expectations to system disposal. The result is a common model and architecture framework, which defines model architecture viewpoints and views, model configuration management, and a model review method.

Karban [ref. 39] describes how the European Southern Observatory (ESO) is fostering reusability by placing common design elements in a parts catalog accessible both inside and outside their user community. The catalog contains the public interface of each reusable element for hardware and software components. This avoids the creation and maintenance costs of identical design elements by every design team in the community. Karban also reports a number of deliverables that are made based on system models: concepts and strategies; system design description (architecture); design conventions, reference design description; different views for suppliers and users; ICDs; cost estimates; and software prototypes.

There are a number of applicable standards and guides for MBSE and document generation embodied in cookbooks and MagicDraw® plugins, distributed under Lesser General Public License (LGPL) at sourceforge.net.

B-2.1.1.4 Lessons learned

Models are meant to be abstractions, so make sure they're useful abstractions: model only as far as you need to answer the question; keep in mind that a model does not have to describe everything in all details, nor does it have to fill in the full space between conceptual and realizational. The Jupiter Europa Orbiter mission model captured high-level concepts and racked up mass from a specific instance [ref. 14].

First description, then analysis. Capture and description are powerful and far-reaching first steps. Just describing something in a formal modeling language like SysML immediately improves communications and understanding; don't underestimate the value of this. Also don't underestimate the difficulty of building meaningful analyses; take that one slow, and don't overpromise. On the Jupiter Europa Orbiter mission, for the mass margin report, even modest ambitions were a bit of a stretch the first time. It took about two work-months to get a working model and report, but the second and third times went much faster. Models were produced representing two additional concepts and mass report; each took only 0.5 work-months; now significant refactorings take just a few days [ref. 14].

Separate the model from the analysis. The Jupiter Europa Orbiter mass analysis achieved a high degree of separation of the model from the analysis, and as a result they were able to run exactly the same mass analysis script on all three mission option models. The more the model can be a self-contained, internally self-consistent and intuitive description of the concept, the more informative it will be. The more the analysis can be separated from the model, the more reusable it will be. As a corollary, one should align the model with the concept, not with the analysis. The Jupiter Europa Orbiter team members initially found themselves adopting modeling patterns that made the analysis scripts easier, but soon found themselves forced to model in more and more nonintuitive ways (drifting back into the Excel® trap). They discovered and adopted the principle that the model should be kept intuitive and aligned with the concept. In the end, they felt the extra work required for smarter analysis tools was well worth the effort [ref. 14].

Keep the focus on engineering products. Tie expectations to project deliverables, not merely modeling solutions. This policy may need to be constantly reinforced [ref. 14]. It has been observed there can be a tendency to try to model everything in a logically complete manner, but the total possible space of what can be expressed about any given system is so large that it quickly becomes unwieldy. Therefore, some criterion is needed to decide what should and should not be in the model, and such a criterion is readily provided by requiring relevance to

specific deliverables in the engineering process. Bayer also mentions selecting model content to answer the questions at hand; thus, it is important to know what those questions are or will be early in the modeling activity. It may eventually come to pass that systems engineering teams use deep and broad system models that are capable of addressing a wide range of possible or even unanticipated deliverables, but for now, these do not seem practical—the state of inheritance from prior workers is still low, necessitating that teams make more narrow choices about model content to address whatever their current questions of interest may be.

Real examples are powerful. It has proven difficult to describe what MBSE looks and feels like to some stakeholders. Actual examples have proven much more effective at conveying understanding and building support. For the Jupiter Europa Orbiter team, the mass model and margin report was the thing that helped the light go on for several skeptical but open-minded stakeholders. Also, the mass model and the margin report were immediately recognized as higher fidelity work than the results of the traditional method. Since parametric cost estimates are based heavily on mass, this is a crucial parameter to estimate accurately. The Jupiter Europa Orbiter team also realized that projects are where the "just do it" happens—working on actual products, where the applications are really worked out—and that is what feeds back into the larger flow of the developing modelcentric practice for others to use. The early examples discover useful patterns that can be fed forward for capture and provision to the next users [ref. 14].

There are usually multiple ways of capturing a system design in a model. The engineering team must work through the different options to determine effective patterns for their situation [ref. 12].

A system model provides a single, unified source of information and reduces the risk of inconsistency or divergence [ref. 12].

Sometimes the source material may contain inconsistencies. This happened on the Soil Moisture Active and Passive (SMAP) project, and the modeling team needed to make assumptions or go back to the project to resolve the inconsistencies [ref. 12].

Capturing off-nominal behavior is not trivial. The SMAP team is currently working on effective patterns to apply [ref. 12].

Having multiple teams (two in the case of SMAP) working on a common system model encourages early agreement on terminology and logical decomposition [ref. 12].

Applications frameworks provide a mechanism to concentrate on domain-specific concepts, and facilitate reuse and documentation of domain-specific knowhow. A significant benefit to using applications frameworks is that the resulting models have independence of platforms and languages, which makes it possible to reuse the models on different platforms [ref. 39].

The prevailing experience in the European Southern Observatory organization is that system modeling is not considered as worthwhile as is recognized already for CAD and finite element analysis (FEA). The reason is that the advantages are not obvious (e.g., traceability, impact analysis, consistency). Moreover, software engineering is often not considered "engineering," and modeling is considered as a software activity. Therefore, it is still necessary to demonstrate the benefits of system modeling; having nice, consistent diagrams for communication is not enough, and additional benefits are needed. These can be shown in the value of:

- Simulation, validation, model transformation
- Reuse of design elements
- Ability to generate multiple artifacts (e.g., documentation, code)
- Model transformations that allow use of capabilities of different tools
- Consistency and correctness across artifacts (e.g., documents, models, products)

Forcing usage of system models is not worthwhile without obvious benefit. A systematic, guiding methodology should be used leading from model to product (e.g., state analysis). The engineering team needs to focus on the engineering product; there is a tendency to get distracted by tool and language problems [ref. 39].

Engineering-specific modeling technology is already ubiquitous, represented in such tools as IBM® Rational® DOORS®, MATLAB®/Simulink®, ANSYS®, SolidWorks®, ZEMAX, BeamWarrior, LabView™, and interface description language (IDL).
These usually require highly rigorous modeling because of executability, strict vendor metamodels, and generating code from the models [ref. 39].

The current language specifications are not perfect. For example, the weak semantics of the Object Management Group (OMG®) specifications inhibit model validation and enforcement of modeling discipline; system modeling with SysML relies on voluntary discipline of the modeler to fill in issues that are left open in the languages. The specifications have to evolve, which requires engagement from community. Keep going, otherwise it is a dead end. There are issues with a lack of tool conformance to specifications and, therefore, of the possibility of exchanging models. In addition, it is necessary to define sound ontologies (both general and domain specific) [ref. 39].

UML/SysML language concepts are now mature enough to be used in practice (as opposed to 2 to 3 years ago). Model transformation languages, checking, and validation are key areas of development, as well as model evolution, organization, comparison, and merging. Without the demonstrated advantages mentioned above, modeling tools would remain only good drawing tools [ref. 39].

B-2.1.2 Program-level Definition of System and Software Interfaces

B-2.1.2.1 Issues

The question of whether models of interfaces can reduce errors and effort in the verification of behavior of external interfaces is being tested by a number of organizations.

Then NASA OCE notes a past inability to share models in a collaborative environment as a problem to be solved, pointing out that development of engineering interfaces, not just the development of interfaces for a project's system of interest, needs attention [ref. 48].

B-2.1.2.2 Assessment

While operational and system performance models are in wide use, different models are needed in addition: interface compatibility and command-and-control flow [ref. 44]. Also observed is the need to develop modeling in the area of aggregated, end-to-end (as opposed to point-to-point) data flows.

The European Space Agency notes the need to allow concepts, properties, and views to drive the conceptual data model in which the system model is expressed [ref. 51]. That is, the mission

concepts, the properties of the system that will be the subject of engineering, and the desired engineering views all require the existence of certain kinds of data in the system model. Therefore, the expectations and content of the data meta-model must be considered as part of the engineering planning and setup process. This is a significant effort at the start of a project, and few standards have yet emerged.

B-2.1.2.3 State of the Practice

Many of the items identified in the design description section are applicable to interfaces as well.

Doyle reports successful high-level definition of guidance, navigation, and control software functional models, including agreement on interfaces with standardizations and associated verifications, for the Space Launch System flight software [ref. 55].

Friedenthal classifies detailed interface models as currently an advanced practice [ref. 17].

The Florida Institute of Technology is currently training students to develop an MBSE flight system model focusing on electrical systems interfaces, with follow-on projects planned to advance into to full project deliverables leading to an eventual launch of a CubeSat; this mission model has already progressed to including high-level interfaces [ref. 56].

The European Southern Observatory is currently producing ICDs based on a system model, among many other deliverables [ref. 39]. To promote reusability, they maintain a parts catalog that contains the public interface of each reusable element for hardware and software components, avoiding the creation and maintenance costs of identical design elements throughout their organization.

The European Space Agency uses functional simulators of system interfaces and end-to-end behavior for design and performance verification, and subsystem and payload verification and validation [ref. 51].

The Jet Propulsion Laboratory's (JPL's) SMAP project is using a system model to describe spacecraft system interfaces in various levels of refinement, ranging from a subsystem-level view down to a low-level view focused on electrical system connections. These are supported by libraries of high-level interfaces, interface types, and low-level interface functions, supported by an ontology implemented as a meta-model for the specification of the electrical system interfaces. They are showing that it is possible to create and maintain the existing engineering artifacts using a model-based approach [ref. 12]. They are also able to quickly establish verifications and validations from the model.

Lockheed Martin is using SysML internal block diagrams and message model elements to maintain interface baselines, to document hardware and software interfaces, and to type system behaviors to interfaces [ref. 15]. Lockheed is using scripts to autogenerate interface software directly from the model.

The Exploration Development Integration Office at NASA is using integrated functional analysis to describe interfaces, off-nominal situations, aborts, hazards, and safety and mission assurance issues, including interfaces for all mission configurations. The Extravehicular Activity (EVA) project is modeling the interfaces on EVA space suits. The integrated power, avionics, and software (iPAS) project is using MBSE to specify interfaces of components in their "Common Vehicle Architecture" catalog. The Orion System Architecture Model includes vehicle interface

and interconnect definitions, although this is being done in an after-design documentation mode rather than during system synthesis [ref. 44].

B-2.1.2.4 Lessons Learned

The Constellation Program encountered problems with a bottom-up design that caused ICDs to focus on low-level details, and there was much duplication across ICDs and interface requirements documents (IRDs). A suggestion for mitigating this situation is to define Program-level, closed-loop state-based behaviors before defining ICDs [ref. 54].

Future Programs and projects will not succeed without initial understanding of computer-aided software engineering (CASE) tool requirements for procedures, standards, and exchange of models at interfaces.

B-2.1.3 Program-Level Design Analysis

B-2.1.3.1 Issues

A constant concern is that of model validity, namely the suitability of a particular model to make the contemplated design decisions and trades [refs. 20 and 57]. NASA possesses a standard for models and simulations that includes methods for credibility analysis [ref. 58].

Modeling should also represent system processes in a time-based model to perform systems performance analysis (capacity/time/resources) [ref. 59].

The NASA MBSE vision has a goal of keeping system designs current with two-way information exchange with discipline models [ref. 48].

Modeling fault conditions and off-nominal scenarios is an area of needed research. The SMAP project found that capturing off-nominal behavior is not trivial; they are still working on effective patterns to apply [ref. 12]. Fluhr has developed a methodology for integrated functional analysis that addresses off-nominal conditions [ref. 44].

A mission timeline that describes key system interactions and how systems interoperate with one another is needed at a high level by an early stage, typically in the MCR timeframe. The mission timeline provides a lower level detail than design reference missions and helps provide insight into the design drivers. A particular issue is consistency between timelines developed by different development organizations; in one recent case, the Constellation Program attempted to develop a detailed (rollup) integrated mission timeline in the PDR timeframe from a number of system-specific timelines, and found various inconsistencies between them. Also, the integrators found that there were not sufficient tools or processes required to create and manage appropriately detailed mission timelines for significant functions such as attitude, power-on, timeline, command, communication, etc. To mitigate this risk, they recommended that future Programs establish a responsible authority that develops and manages a master mission timeline (or sets of timeline data) to be used by all of the projects as the authoritative timeline to be used in analysis. Specific projects or organizations could be delegated population of the authoritative data within the timeline (i.e., development of the trajectory timeline). Additionally, this authority would develop appropriate tools allowing access of the master timeline data to all Program participants, along with interchange mechanisms to allow for these data to be exchanged with appropriate models/analysis tools. This capability should include an integrated activities modeling capability, specifically a discrete event simulation like the JSC Mission Operations Directorate was developing at the time [ref. 60].

The European Space Agency has been pursuing the capability for concurrent engineering in a multidisciplinary environment, with the goals of integrating customer, engineering team, tools, project data, mission and system models, and enabling simultaneous participation of all mission domains, including cost engineering, risk analysis, programmatics, operations, CAD, and simulation. Besides the organizational and information technology challenges, they also have noticed a change in the overall engineering situation between the early and late design phases. In the early phases, the situation is characterized by tens of users, one or few organizations, near real-time collaboration in minutes or hours, trade studies and strawman models, requirements in a state of flux, the need to support decision making and program formulation. In contrast, the later phases are characterized by hundreds of users, tens of organizations, synchronization times in days or weeks, formal configuration and version control, a strict requirements baseline, formal detailed verification and validation, and orders of magnitude more models/data [ref. 51]. Thus, it is necessary that the engineering environment adapt to the change in situation between earlier and later stages.

The engineering environment also should recognize that there are at least two broad types of analyses to be carried out in the design process: static analysis, which is concerned with component and interfaces specification, and dynamic analysis, which is concerned with the response of the system to stimuli [ref. 44].

B-2.1.3.2 Assessment

The INCOSE has identified six leading MBSE technologies that can be used for Program-level design analysis:

- IBM® Telelogic® Harmony®-SE
- INCOSE Object-oriented Systems Engineering Method (OOSEM)
- IBM® Rational® Unified Process for Systems Engineering (RUP SE) for Model-Driven Systems Development (MDSD)
- Vitech MBSE methodology
- JPL State Analysis (SA)
- Dori's object-process methodology (OPM)

They also provide: references to Embedded Computer System Analysis and Modeling (ECSAM), Model-Based [System] Architecture and Software Engineering (MBASE) embedded and software-intensive methodologies, an acknowledgement/overview of Wymore's mathematical foundation of MBSE, a status and update to OMG Executable UML Foundation, and a reference to Cloutier's work on model-driven architecture (MDA) for systems engineering [ref. 61].

B-2.1.3.3 State of the Practice

Jody Fluhr, in Moody [ref. 44], reports that the NASA Exploration Development Integration Office, Tri-Program (21st Century Ground Systems Program (21 CGSP), Multi-Purpose Crew Vehicle (MPCV), Space Launch System (SLS)) Integration Office has an active Integrated Mission Analysis (IMA) activity, which is using extensive functional modeling of a major system-of-systems (SoS). The IMA is designed to identify the needed system capabilities, starting first by modeling the design reference missions in terms of phases, segments, and activities. These are used to identify required system capabilities. The system capabilities are then linked to the mission phases, segments, and activities to show the capabilities needed at

various times. The types of capabilities considered include internal system capabilities, system-to-system interfaces, and mission configurations. From these, Fluhr models a conceptual systems-architecture system captured from the concept of operations; systems are linked to the mission model to indicate which systems are required and when. Also, systems are linked to capabilities to establish allocations. Mission configurations are identified and linked to the mission model to show where they are applicable, and capabilities are linked to mission configurations to show applicability. The result is thorough traceability of the mission needs to the system characteristics, which can then be extended to enable integrated functional analysis (IFA), requirements development, interface development, and an integrated design definition document. Fluhr also describes IFA as modeling interfaces, off-nominal operations, aborts, hazards, and safety and mission assurance analysis.

Othon, in Moody [ref. 44], reports that the NASA MPCV is described in the Orion SAM, which covers avionics, power, and wiring functional modeling. The SAM is an analytic model representing vehicle function, structure, and behavior. The models of vehicle function are system capabilities without operational context, while the models of vehicle behavior are functions in their mission context. The models of vehicle structure include system and component hierarchies, as well as interface/interconnect definition. The SAM contains operational behavior in terms of use cases, activities, and interfaces; a physical architecture in terms of modules, subsystems, components, and interfaces; and a functional decomposition in terms of use cases, states, and operations. These are all related externally to requirements, operations concepts, architecture descriptions, operational timelines, a master equipment list, and a block upgrade plan. A large number of design requirements document artifacts are produced from the model. The SAM is not part of the system development cycle, it is a documentation of the as-designed system and follows the design activity.

Othon also reports on an iPAS project. iPAS is an intermediate element in a hierarchical engineering management strategy with domain labs at the bottom, federated labs integrating the domain labs, iPAS capturing design successes from the federated labs and identifying common architectures that are expressed in a common vehicle architecture (CVA) catalog, culminating in vehicles that use those common architectures.

iPAS provides a standardized environment for hardware/software evaluation and test, in a world of multiple, parallel development tasks. There are two elements to iPAS: vehicle hardware and software (the "iron bird"), and common test services and products (the "iron nest"). iPAS also develops the CVA catalog to capture products from successful project or research and development efforts. They are using MBSE to communicate elements of the catalog, in terms of component identification, interface specifications, and performance models.

B-2.1.3.4 Lessons Learned

Bayer *et al.* [ref. 14] recommend the following practices to be successful with high-level design analysis: separate the model from the analysis so that as needs for new kinds of analysis appear, the model can continue to be useful; and keep the focus on engineering products so that the tendency for analysis to expand beyond real-world concerns can be mitigated. Also, Bayer *et al.* have found that real examples are powerful for showing the value of a model-based approach.

B-2.1.3.5 Challenges

There is a need to demonstrate better where gains can be made in correctness, consistency, time, and money.

Currently there are still a large number of working products that are produced from the existing habits of models hidden within the engineer's mind, with model-based approaches relegated to after-the-design description. It is necessary to make progress on delivering more of the working products based on explicit models before the end of the process.

There do not yet seem to be any ready examples of practical work that include integration of system models, control engineering models, and control software models with SysML based on a state analysis paradigm, although this area is mentioned by Karban [ref. 39] as an area of future work for the European Southern Observatory. Given the prominence of these areas in most of the methodologies identified by INCOSE, this seems a natural area for future emphasis for NASA.

B-2.1.4 Modeling Integrated Performance

B-2.1.4.1 Issues

Currently, the focus in MBE has been primarily on discipline-specific models. The capability to support integration across discipline lines tends to be limited or missing, and when there are integrations, they tend to be "point-to-point" solutions that lack the generality that would make them useful downstream in other situations. Yet, performing integrated analysis at the SoS level, for typically complicated system scenarios such as an aborted launch, seems to inevitably lead to a need for interactions between system models, software models, discipline models, and integrated timelines. There is a need for a cross-discipline integrated framework for concept, design, and analysis of systems based on standards, open architecture, and existing commercial-off-the-shelf (COTS) tool sets.

B-2.1.4.2 State of the Practice

Oster [ref. 15] describes a campaign at Lockheed Martin to integrate a wide range of disciplines and phases of the system development life cycle under the concept of a "digital tapestry." Their view is that a well-defined SAM is the loom that weaves the many threads of digital information together from different disciplines. By viewing the SAM as the hub of the digital tapestry, an integration pattern emerges enabling cross-domain connectivity with a minimal set of required integrations. Lockheed allows the center of integration to change during the life cycle. During the concept refinement and design/development phase, they use the SAM to integrate system requirements, system test, system cost, software, firmware, electrical, and system analysis. Prior to that, during the concept development phase, they use the operations concept for integration. Later, during the engineering and manufacturing development phase, they use a three-dimensional CAD model for integration. The operations and sustainment phase is still an area of development for them, and they are planning to have an integration center for that phase of the life cycle.

A key issue associated with integrated performance modeling is the question of how much to model. Somervill and Johnston [ref. 19] are investigating using risk to drive key modeling decisions. Their plan is that top risks will be used to guide which areas of the SysML model should be the most thoroughly developed. Risks will be shown on the SysML diagrams as they

are related to elements. Risk mitigations that involve changes to the system design or processes will be documented in the model (e.g., design evolutions, activity diagrams for mitigation plans).

Vera *et al.* [ref. 20] describe a current capability that pulls together SLS functions (modeled in Enterprise Architect (EA)) to the Exploration Systems Directorate capabilities (modeled in Cradle) to create an up-to-date IFA report. Also, Vera *et al.* [ref. 20] report a project to demonstrate a System for Tracking Operational Readiness for Missions (STORM) at the Ames Research Center (ARC) OCE. STORM is a multisystem integration to support capturing engineering content and story in support of reviews that integrates Windchill®, Xerox® NX, and the ARC Risk Information System.

Additionally, Vera *et al.* report that a group of entry, descent, and landing discipline experts have developed an integrated multidisciplinary design optimization process referred to as the Co-optimization Blunt Body Reentry Analysis. This process assesses planetary atmospheric entry system concepts that account for shape, trajectory, thermal protection system, and vehicle closure. Specific disciplines included are aerodynamics, aeroheating, thermal protection systems (ablative, non-ablative, flexible), structures, and trajectory

B-2.1.4.3 Challenges

Integration success requires a good understanding of the requirements for and limitations of integrating the separate models and their interfaces. There is a challenging scope and variety of integrated models within NASA:

Domains: ascent, descent, landing, roving, aerial flight, orbit, rendezvous, proximity operations, docking
Applications: detailed system performance assessment, flight-software/algorithm verification, flight software data loads, real-time operations, dispersion/failure analysis, reconstruction
Levels: spacecraft to mission; multilevel fidelity

B-2.1.5 Maturing Model-based Products and Activities in a Collaborative Environment

B-2.1.5.1 Issues

A key challenge in creating a model-centric enterprise is the integration of multiple modeling domains across life cycle and across the full scale of the engineering enterprise, from concept to sustainment, and from system to component [ref. 48]. Furthermore, the multidisciplinary model-based system design environment needs a tool set that spans the engineering disciplines to integrate a full range of system engineering responsibilities: requirements decomposition, design, development, and implementation [ref. 19].

Bromley [ref. 48] advocates a reusable models repository and standard templates for common component and subsystem building blocks. Having these would allow rapid model generation, which is particularly useful in early life-cycle engineering trades.

Bromley also points out that a real model-centric enterprise requires software interfaces between disparate tools.

There is a need for a configuration-managed, controlled model repository. Here the concept is that of model configuration management, distinguished from system configuration management. The challenge here is to provide a lightweight versioning and distribution capability to facilitate access to the available models.

In the area of model interoperability, Jenkins [ref. 16] has identified a list of model exchanges that exist now and ones the community wishes it had. He found widespread interest and multiple implementations of SysML-to-document transforms, particularly using DocBook. He also found a high level of interest in:

- Transformations for import/export/synchronization among system modeling tools
- Integrating discipline-specific tools
- Integrating system modeling with general computational services
- Integrating system modeling with high-end simulation/analysis tools
- Integrating system modeling with specialized correctness checkers/provers

A significant issue is that of model integrity—the condition of models with respect to being complete, whole, consistent, and adhering to the codes of quality or conduct that are important to any particular enterprise. Presently, no widely accepted measurement rules for this important characteristic exists, though there have been a number of auditing policies applied over the last few years that have proved useful for improving the integrity of models [e.g., ref. 37]. It is certainly possible to measure completeness compared with an external expectation of content, compliance with modeling guidelines, language compliance, and to detect errors such as mismatched interfaces or violated assumptions about the surroundings.

Measurement of model integrity as a prerequisite for, or as a part of, a review process is an important check to make before relying on models for derivative work such as continued design elaboration or decisions about the condition of the engineering process. A qualitative (with some quantitative components) method for evaluating the credibility of models and simulations exists in the NASA Modeling and Simulation Standard [ref. 62], but it applies to decision conditions, risks, and evidence about the models rather than direct measurements of the condition of the models.

Another significant issue is that there are times, especially early in a system design life cycle, in which there are large chunks of missing data, or there are episodes of large changes, and frequent additions of new concepts, etc. It is not clear how model integrity is established under such conditions, though it is clear that there can be "zones of integrity" where the model can be expected to be good based on the information available. However, some as-yet-unkown mechanism is needed to deal with the boundary of the zone of integrity and to discover ramifications when new concepts or information are introduced to the model.

Maintaining integrity as the design evolves is also an issue. In the ideal, this would be a continuous process that is carried out when models are changed, before check in, before review, before baseline, etc.

When sharing information among multiple organizations, a frequent challenge that arises is that of protecting sensitive information. Access must be arranged to be consistent with both the required restrictions, as well as the duty to share information needed to perform the engineering and management work. Coupled with this concern is that of providing appropriate marking of information, which may not be expressed in the usual paper form, being purely electronic in nature.

Finally, integrity of tool interchange is a concern and a needed area of development. It is fairly common for the information that is strictly expressible in a given language (such as SysML) to pass from one tool to another without corruption if both tools work in the same underlying

language and produce views from the same technical viewpoints. If they do not, however, there is a challenge to translate between different languages and viewpoints. Furthermore, informal information in a model, such as positions of objects in a view, colors and graphic symbols, and extensions beyond formal language construct, usually do not transport correctly. The current situation is that such information has to be handled through some side channel, typically a labor-intensive one.

B-2.1.5.2 State of the Practice

There is a wide variety of tools available for collaboration. In 2012, Bromley *et al.* [ref. 63] surveyed six large industrial concerns, and Jenkins [ref. 16] conducted an informal survey at a recent MBSE workshop with NASA, industrial, and academic participation. The aggregated list of tools in use that they discovered appears in Table B-2-1.

Table B-2-1. Tools for Model-Based Products and Activities in a Collaborative Environment

Genre	Tool
Product Data Lifecycle Management	Creo® ELEMENTS/PRO® Creo® ELEMENTS/PRO® Wildfire Primavera® TeamCenter ePIC TeamCenter Unified Windchill®
Computer-aided Design	AutoCAD® CATIA® I-DEAs Mentor Graphics® ModeFRONTIER® Xerox® NX Pro/E® SolidWorks® TeamCenter Unified Zuken™ E3 Zuken™ ECAD
MBSE	DOORS® Sparx EA® MagicDraw® ModeFRONTIER® Modelica® Phoenix® Integration Pro-STEP RequisitePro® Rhapsody® SEER®-SEM Teamcenter SE
Modeling and Simulation	ANSYS® Delmia® Design Pro Dymola®

Genre	Tool
	LabVIEW™ Mathcad® MATLAB® Simulation Toolkit™ Simulink® STK Systems Tool Kit® Thermal Desktop®
General	Apache® SCXML engine (execution of models) Atlas Transformation Language (ATL) (transformation language) Cameo™ Inter-Op (model exchange) DataHub (data integration) Eclipse™ Modeling Framework (modeling framework and code generation facility) Java™ (object-oriented programming) Java™ Pathfinder (model checking) JT (visualization format) M2T Tools (model to textual artifacts) MOFLON (model transformation, code generation, testing) OMG® Query/View/Transformation (QVT) (transformation language) Python™ (scripting, object-oriented programming, general programming) Ruby (object-oriented programming) State Chart Extensible Markup Language (SCXML) (modeling language) Systems Modeling Language (SysML) (modeling language) Teamwork Server (data integration) Unified Modeling Language (UML) (modeling language) XML (interchange format)

B-2.1.5.3 Challenges

In general, a lack of tools exists for integrating system modeling capabilities across domains, although there are recent developments occurring in InterCAX SLIM and 3DS Simulia®.

B-2.1.5.4 Lessons Learned

Tools must come with (free) viewers to ensure the widest collaboration possible, without requiring the model viewer to invest in the heavy tools of the model providers. This is important when dealing with small businesses [ref. 63].

De-scoping MBSE tools and environments can cause some failures. It is better to start small and then expand.

It is preferable to minimize tool customization (i.e., do little of, for example, Java™ or C compilation) and maximize tool configuration (i.e., do lots of, for example, XML, parameter changes).

B-2.1.6 Model-based Acquisition

B-2.1.6.1 Issues

In contractual insight/oversight, design cognizance, and Program-level integration require two different types of relationships with the contractor. The former requires an open, exploratory atmosphere, while the latter requires strictness and accountability. These two roles may produce conflicts of interest and need to be separated [ref. 29].

From the contractor perspective, any given contractor may have many government customers all with different needs and standards. NASA and other Agencies should standardize more on model-based deliveries. This way, several Agencies benefit, and the contractor can move more effectively in cutting the cost of deliverables.

There is a general issue with access limitations imposed legislatively or contractually. This becomes significant if all the information is contained within a model; however, there are partners who must have access to the model, but whose access must be limited.

Abstraction and subsetting from models is a key to mitigating the access problem. It is necessary for an integrator to identify what output from each model is important to the integration activity. This is about getting the model interfaces "right." Then, one can abstract the inner workings of a model (especially if it contains the vendor intellectual property) and still use the model to the fullest extent by exercising its interfaces for integration purposes.

Watson [ref. 57] recently held a discussion on the subject of model-based acquisition with model-based practitioners and identified the following issues:

- Contractual clauses/expectations for model exchange (reuse, neutral format, intellectual property rights)
- Data consistency, how to verify data transfer, export/import consistency, Contract Data Requirements List (CDRL) definitions/expectations
- Model artifacts as requirements for CDRL delivery
- How is a model viewed? Legally? In intellectual property terms? What information is required for delivery?

The discussion group made the following recommendations:

- Set up guidelines for contracting addressing format of delivery, expectations of CDRLs, intellectual property/International Traffic in Arms (ITAR)/etc. considerations.
- Investigate how the Department of Defense (DoD) has specified DODAF-based delivery
- Investigate how to specify clear acceptance criteria for delivery of product

B-2.1.6.2 State of the Practice

There is little, if any, evidence within NASA of model-based acquisition. This may be an indication that the area is a potential growth area, likely to yield gains comparable to what has occurred within individual disciplines as they have adopted model-based techniques.

B-2.2 Supporting Program Integration - Integration, Verification, and Validation

B-2.2.1 Issues

Expectations of early-stage integration, verification, and validation must be adjusted to take into account the limited fidelity that is available in those phases. Design domains will not all be at the same degree of maturity, and the fidelity will be limited by maturity at different tiers of the design. It is not reasonable to expect fine details to be accessible where maturity is low. However, it is reasonable to expect early-stage work to detect large-scale gaps or blunders, missed goals, previously unknown dependencies or constraints between elements, and undesired emergent characteristics arising either from inadequacies in the requirements, or from the early, large-scale choices. Even so, if large-scale problems exist but are the result of low-level details of systems, they may not become apparent until low-level details are defined. This limitation may justify exploratory definition of fine details using models, depending on the risks involved.

Expectations of late-stage integration, verification, and validation can be more expansive. It is reasonable to expect the process to detect gaps and blunders at all scales, and to ask the process to perform when resources (people, facilities, test articles, time) are cut short by overruns or schedule slips. Here, machine aids can help extend the resources available or make more efficient use of them. An issue arises at this stage in the necessity to have complete models that are fully consistent with the actual systems as designed and built.

In both early and late stages, it is reasonable to expect the process to provide clear, credible explanations of gaps and blunders, to provide definite corrective action recommendations, and to maintain a light footprint that promotes productivity of the organization.

B-2.2.2 State of the Practice

The current state of the art in MBE makes it practical to use virtual integration, a technique described later in the section on Methods and Practices. Stoewer [ref. 33] reports that simulations are currently used to test models to verify performance and functionality of the system of interest. Models are verified against real components and software in tests. Large numbers of validated tools are available in the fields of structures, thermal, Attitude and Orbit Control System, communication subsystems, etc. Importantly, top down or system models with some integration capabilities are available in early tool sets in the fields of architecting, requirements management, etc., and life-cycle time and process models are available for development, testing, manufacturing, etc. Kahn and Standley, in Ingham [ref. 34], report that behavior diagrams of limited subsets of system behavior are contributing to the rapid discovery of scenario-relevant interactions and interfaces for verification planning. Simple transformations of actors between operational and test scenarios are facilitating the rapid preparation of test procedures, and test configurations and context can be captured in a support equipment catalog. Requirement verification can be traced to elements in the model; by querying the model for requirement summaries, it is possible to handle requirements in a more complete and less cumbersome manner using views that are easier to interpret.

DeLaurentis [ref. 36] points out that agent-based modeling can be used to discover emergent behaviors, both desired and undesired. The Defense Advanced Research Projects Agency META program is exploring complexity across layers of design abstraction, seeking to understand multi-modal (information, force, energy, matter) interactions as designed and

interactions that are off-design or undesigned. That project is exploring stochastic formal verification; the project currently can relate it to a real-life problem like network performance, for example, fault occurrences or capabilities of assets.

B-2.2.3 Lessons Learned

Integration decision making can be facilitated using models. For example, in the case of a Certification of Flight Readiness (CoFR), a program-level CoFR calls for a single "yes" or "no" from Systems Engineering and Integration (SE&I) for certification. The products and sub-decisions needed to reach that decision can be decomposed according to the top-level functions, mission capabilities, and the timeline contained in a system model. Questions detailing criteria concerning readiness for flight can be developed, and then be evaluated using criteria judging readiness of the system, suitability of the mission design, residual risk, etc. A practical plan for deciding readiness can be derived from the model [ref. 35].

Morillo [ref. 64] has noted that access to information needed for integration can be problematic. In the case of the Constellation Program, for example, access to many technical repositories was overly constrained. The Program and projects established many compartmentalized sites for access to various tools, data directories, and technical review repositories all with their local access rules. The detriments were that individuals had to apply for access to each separate site, often by providing redundant information and having to wait sometimes for several days or longer to gain access. In some instances, leads and managers had to interfere. Delays became crucial when there was a limited time window (typically 15 days) for document reviews and comments. An inefficient workaround had to be put in place to mitigate lack of access to these primary sites. Integration organizations had to build mirror sites where copies of documents were downloaded as a way to ensure that all reviewers had timely access to their needed documents. Because the tasks of program technical integration rely on timely access to information from the various interfaces, overly constrained and limited information access made the integration job more cumbersome and less productive. (For example, when a reviewer had no access, review comments had to be handed to third parties with proper access, resulting in further inefficiencies and potential technical errors.) Several reasons were cited for the compartmentalizing access to information such as ITAR/sensitive but unclassified (SBU) data, intellectual property concerns, preliminary data not ready for wide distribution, etc. These reasons were and continue to be valid. However, the implementation can be much improved and better streamlined; not doing so leads to inefficient informational barriers that carry a severe cost and risk impact. It was recommended that access processes be streamlined, for example, filling out a personal profile once (with personal data, employment, citizenship, organization, affiliation, task/responsibility, etc.). Access to sites and directories would be based largely on that profile, with only a few exceptions requiring special handling.

Congruence between reality and information available is critical. It has long been known that computer models have hidden flaws, not the least of which is poor input data [ref. 65].

B-2.2.4 Challenges

With integration, there is a significant challenge in that systems may be designed top down or bottom up. The maturity of the models at any point in time may be higher in lower levels, or vice versa. This makes it desirable to find methods of dealing with incomplete information but still be able to make inferences about integration questions (with some degree of risk left pending until accurate information becomes available).

B-3.0 Objectives during the Program or Project Life Cycle

Given that there are a number of software- and software interface-specific issues in SoS development, and a relatively low level of specificity of actions required to address those issues within NASA standards, it will be helpful to identify specific actions to address software interface issues for each phase of the system development life cycle. This section presents a necessary set of objectives to be achieved for a large, complex SoS software interface situation. On smaller efforts, some of the issues may be less troublesome and be satisfactorily handled with simple or no particular treatment of the issues; in these cases, the objectives should be tailored accordingly.

The set of life-cycle objectives presented here should be used as a guideline for planning, assessment, or diagnosis of software interface management efforts. This section is organized according to life-cycle phase, with the intent that it will be fully integrated with the overall Program or project control and engineering effort.

B-3.1 Concept Studies

B-3.1.1 Discover and Adjust the System Boundary

Dividing a system of interest from its environment during the concept study phase is crucial to the success of the system development. Indeed, the division governs the definition of success; whatever ends up inside the system boundary will eventually be judged as to whether it meets the need within the range of cost, schedule, and risk considered acceptable. Whatever is inside the system will be considered to be under the control of the project managers and system engineers; whatever is outside the system will be considered at best to be somewhat amenable to the influence of project managers and system engineers, or perhaps completely fixed and unchangeable—something for which the manager and engineer will never be called to account. From the standpoint of determining success and failure, it is essential that the influences needed to achieve success are available to the manager and engineer, while at the same time it is essential that accountability for outcomes excludes that which cannot be controlled.

The entire notion of system versus environment requires making a distinction between what is and is not inside the system. This distinction can be well represented by a combination of physical and nonphysical models. For software interface work, sufficient information must go into such models so that it is possible to distinguish between what is inside and what is outside the system from the software viewpoint.

B-3.1.1.1 Obtain a Black-Box Model of the System

Black-box interactions can be defined between a system and its environment or between system timelines. Black box means that the internal behavior is hidden and the external behavior is visible. Ultimately, interface requirements should be derived from black-box interactions or black-box system requirements.

It is critical, for software interfaces, to obtain a black-box model of the system in relation to its environment during the conceptual phase. For the physical part of the model, the connections relevant to software should be identified. These may be communications channels, networks, server environments, or user platforms. There may also be physical issues that the software is to manage, such as accuracy of targeting, quality of products emerging from the system, or physical

conditions to which the system will be expected to adapt; these should appear in the black-box model.

For the functional part of the black-box model, the functions relevant to software should be identified. These may be transformations, storage and retrieval, adaptation to the environment, or adaptation to changes in the system including fault management.

Understandably, at first the information in the conceptual models will be imprecise and incomplete. A ramification arises from this situation: as precision and completeness improves, it may be necessary to correct the system boundary and success criteria.

It is fairly common that the black-box phase of system definition is truncated or avoided altogether during conceptual study, going directly to a white-box description of the relationships between interior system elements. Modeling that ensues may focus on internal concerns, which in itself needs to be done eventually but may become a trap if done too early. The trap arises when the complexity of modeling the interior concerns fills up the minds of external parties and the available time for discussion with them, to a degree that it crowds out the modeling that needs to be done to discover the needed relationships between the system (as a black box) and the environment. Worse, the existence of detailed internal modeling may give the impression that a high degree of understanding has been reached, suppressing awareness that understanding of the system in relation to its environment may still be quite poor.

B-3.1.1.2 Evaluate Differences between Hardware and Software Boundaries

There is often a difference between the location of the hardware and software system boundary. These should be evaluated as to whether the difference is needed, and if it is, the difference should be carefully propagated through the rest of the system design.

> Example: a Web-based interface may have the Web page interface at a server interior to the hardware system boundary (interior network, routers, interior computer, interior operating system, or middleware intervene between interior software interface and exterior hardware interface).
>
> Example: a client-server architecture may have a client residing in hardware owned by the environment, exterior to the system's hardware boundary (network, routers, exterior computer, exterior operating system, or middle ware intervene between exterior software interface and exterior hardware interface).

Models that include both physical and functional boundaries can be an excellent tool for managing decisions about where the software boundaries should be placed.

B-3.1.1.3 Identify things that will interact at the software interfaces

Recall that a software interface is a place where nonmaterial things meet. Therefore, describing the system boundary well enough for software interface work entails identifying something nonmaterial about the system that will be interacting with the environment. There are a great many possibilities for what types of nonmaterial things might be used for this purpose. Some specific types are:

- Functions that will be interacting with the environment, for example, "The weather forecasting function will receive meteorological data from the environment," or "The security management function will provide authentication to the environment."

- Information that will be exchanged with the environment, for example, "The environment will supply the target location," or "Transaction control information will be shared between the environment and the system."
- States that will be managed by the system or that exist in the environment, for example, "The system will transition from the state of pad operations to launch operation when authorized by the environment," or "The system will guarantee that an interacting party in the environment is kept in a safe state."
- Behaviors or activities or roles involved in interactions with the environment, for example, "When the environment is too windy, the system will feather the turbine blades," "The system will switch to internal power when the environment disconnects the power line," or "The environment will be the master and the system will be the slave."
- Rules of system or environment operation, for example, "The system will direct debris away from population centers," "The system will not initiate trades after the trading period is closed for the day," or "The environment will continuously attempt to deposit malware within the system."
- Services or capabilities or ports that the system is expected to provide or to use, for example, storage, computation, libraries, Web sites, communications, or control ports.
- Conditions or desired effects or measures that the system is expected to achieve or cope with, for example, system modes, system failures, quality standards, or control goals.
- Conceptual representations of significant entities—classes of objects that will eventually be instantiated in either the environment or the system—that are involved in interactions between the environment and the system, for example, "The server will receive files deposited by users in the environment," or "The database in the environment will respond to queries from the system." These conceptual entities are representations of actual entities that will eventually be defined in the system design and realization process, that define the conceptual space in which the system boundary exists for software purposes. Usually the information available for such representations will be approximate, in need of refinement during the system development process. However, in cases where the environment or the sub-elements of the system are already defined, the precision and accuracy of the representations may be high.

In any particular system development, there will likely be specialized software concepts unique to the system under consideration. These require careful description and development of relationships to generally understood concepts.

B-3.1.1.4 Check for Interface Pathologies

The system boundary may need to be adjusted to avoid pathologies. One pathology is to place a software interface at a boundary where a high rate of data exchange is needed.

> Example: In instrument pointing, the job of deciding where to point the instrument may be given to the environment. The pointing information could be expressed in fine steering commands at a rate of 1 kHz or more, assigning to the environment the function of converting from a desired target coordinates into fast instrument commands. Alternatively, the system boundary could be adjusted so that the pointing information is expressed in gross inertial commands at a rate of one per observation, assigning to the system the function of converting to fast instrument commands.

Another pathology is to place the software interface at a functional boundary where excessive information about the system internals is needed for the environment to perform its functions.

> Example: In spacecraft timing, an onboard clock could be driven by a free-running oscillator subject to aging, thermal, radiation, and acceleration drifts; the environment could be given the function of monitoring, correcting, and forecasting clock adjustments so that commands can be coordinated accurately with an external timescale. This necessitates the environment accessing aging, thermal, radiation, and acceleration information, both histories and forecasts, thereby creating extra complexity in both the environment and interfaces to the system. Alternatively, the system boundary could be adjusted so that an external time reference is supplied to the system, and the system assigned the function of disciplining its onboard clock to match the external reference. This example points out that system models can be important for managing the software functions between the environment and the system in both the pathological and the improved interface locations, but the complexity and cost of the associated models can be drastically different. Specifically, in the pathological interface, the models require detailed time-sensitive information about the internal characteristics of the system and must be constantly maintained over the system lifetime at considerable operational expense. The improved interface location, on the other hand, needs only approximate modeling information about the likely range of effects. This type of information is sufficient to verify that the system internal timing error can be made sufficiently small given the update rate to the external time reference; this verification model need only be maintained during the design process, avoiding all the operational costs of maintaining the other model. More will be said about using models for verification later; the main point here is that choosing a different system functional boundary can improve the software interfaces.

> Example: In data communication, aligning the software boundary at a physical boundary, for example, bits on a wire leaving the system, may require the system to communicate externally with external operators about excessive detail of communication such as bit rate, modulation type, signal strengths, missing messages, etc. Alternatively, the software interface could be placed at a higher layer, relying on lower level protocols and common modems to negotiate adequate communication, thereby simplifying operational details to the level of "this email got through, or it did not get through." This is the strategy currently used for commodity communications; some may remember the days before negotiating protocols were embedded in modems, when all communications parameters had to be set manually on both ends of the interface, and the attendant problems that resulted. The key difference between the two approaches is in where the models of the communicating parties are kept, and who has to be involved in keeping them synchronized. This illustrates a principle that the location of models—not only about the system, but within the system—can be a significant software interface issue.

A significant pathology can arise from overly strict alignment of software and hardware boundaries. If they are forced to coincide, it is unlikely that characteristics of good interfaces will be arranged for both hardware and software.

> Example: Suppose that there is a communication boundary between two systems at a simple hardware interface, say, a wire or a radio link. The complexity of the hardware interface is low—simply a connector or an antenna—and the systems are well decoupled

in a hardware sense. However, the software interfaces become quite complex because the two interfacing parties must agree on a wide range of details at many levels. This situation creates a need for a side channel to communicate those details between the development organizations, typically an expensive one because of the labor involved. A superior solution is to allow the software interface to exist at a different "location" than the hardware interface, at a higher plane internal to the software of each system. Then, a single provider can be appointed to create elements that move information between the two interface planes in the communicating systems, via the software drivers, the hardware, and the hardware interface. This, by the way, is the concept behind the use of common a modem with its driver software.

A serious pathology can arise if there are unrecognized high-level software systems. This can occur if partitioning proceeds to hardware without going through the step of functional allocation to hardware and software in the top-level system. The owner of the top-level system is then left without any awareness of a connection between the top-level goals of the system and the requirements on software of the lower level elements. Correspondingly, the owner of lower level elements is unaware of what needs to be done to achieve the top-level goals. And, being unaware, the parties will experience an insidious perception that there is nothing to be done about software interfaces, when in fact there is.

Example: A communication system has the job of moving information from one system to another. Faults may occur in the lower elements. If there is not a fault management function identified for the top-level system, the requirements for lower-level elements will not contain functions to manage faults. As a result, it is unlikely that goals for completeness of information transfer will be met.

Example: A space transportation system has the job of moving people safely from one point to another. This entails a decision as to where and when is safe to travel. If there is not a safety decision function identified for the top-level system, the requirements for lower-level elements will not contain functions to make such decisions, or to provide information needed for decisions to be made by other elements. As a result, there will either be no system to make safety decisions, or a *sub rosa* system will be created with attendant difficulties in performance, quality, integration, verification, and validation.

A useful set of heuristics for aggregating and partitioning has been identified by Rechtin, and then extended by Maier and Rechtin [ref. 26, pp. 273–274]. Potential pathologies covered by their heuristics are:

- Grouping together elements that are unrelated
- Drawing subsystem interfaces that prevent independent implementation of the subsystems
- Choosing configurations that increase communication between subsystems
- Choosing elements so that they depend on each other, with higher external complexity than internal
- Aggregating systems that have a conflict of interest
- Aggregating so as to produce untestable subunits
- Having too many elements (moving complexity from the elements to the interfaces)
- Slicing through regions of high rate of information exchange
- Having a system structure that does not resemble the functional structure
- Mismatch of interfaces to higher levels or harder-to-change systems (such as humans)

- Considering only factors within the system boundary
- Ignoring interfaces created by reality
- Increasing the time individuals spend interfacing

Maier and Rechtin [ref. 26] offer solutions for each of these pathologies.

B-3.1.1.5 Use models to discover and adjust the system boundary

Modeling can assist in discovery and adjustment of the system boundary. For example, Friedenthal, Moore, and Steiner [ref. 3] describe the use of SysML to precisely define the system context using blocks, ports, and interfaces.

SysML can also be used to efficiently represent nested structures and connectors, describing whatever details of a structure are known or decided at this early stage. For example, it is possible to express a connection directly between a lower tier entity, such as a tire, and an external entity, such as the road, without having to create fictitious interfaces to connect the road to the vehicle and then down through the vehicle's hierarchy of decomposition: vehicle to suspension, suspension to wheel, and wheel to tire. Applying this concept to software interfaces, SysML allows connections directly to a particular interface port to be expressed, without creating a fictitious connection to the top-level application, or a fictitious series of interfaces from the top-level application through its internal hierarchy of components down to the particular interface port. This is a distinct advantage in that it preserves information necessary for subsequent design and verification activities, without burdening higher level views with low-level details. One case was observed where the lack of this advantage had program-wide impacts. Specifically, the Constellation Program Office encountered difficulty with nested interfaces at the top level and chose to exclude them, delegating their management to lower level entities. The resulting loss of information in the high-level models ultimately made it impossible to relate the lower level interfaces to higher level requirements [ref. 11].

As a practical matter, inter-comparisons of different system context models made by the owners of the system of interest and those made by owners of the surrounding systems in the system-of-interest environment can be effective in revealing gaps and inconsistencies in the understanding of the system boundary and in revealing opportunities for adjustment to improve either the system of interest or the systems in the environment. When the contexts are expressed in machine-accessible models, they can be subjected to machine-assisted audits and model checking to reveal misunderstandings between the system owners.

B-3.1.2 Define the Problems to be Solved – Expectations, Emergent Characteristics

After defining the system boundary, it is necessary to define the expected performance of the functions within the system: how well, how fast, how much, how often. During the conceptual phase, these may need to be expressed in terms of desired effects on the environment rather than characteristics of internal functions.

It is also necessary to define the expected quality of the solution: how often can it fail or what guarantees are required from the existence or operation of the system.

Constraints need to be defined: When the solution must be ready, what are permissible and impermissible interactions with the environment, which types of system content are permissible and impermissible.

Defining the emergent characteristics is critical: what is expected from the system that does not result from just having a collection of elements? Without this information, there is no reason for having a higher level system, and it becomes difficult to do any engineering that relates high-level desires to software interfaces.

These factors set the stage for the top-level software system design and for defining the software interface requirements. The problem to be solved, expectations, and emergent characteristics should be captured in a model so that they can be traced easily to the software design and interface requirements later on.

For software work, it is important that sufficient information regarding the overall expectations is obtained to make it possible to discern what is expected of the system from the software viewpoint. Besides being a necessary base for deductive reasoning about what the software design should be, the information is a necessary base for software acceptance decisions later on. Without such information, it will ultimately be impossible to decide on rational grounds whether the finished software is acceptable.

Modeling can assist in defining the problem to be solved. For example, Friedenthal, Moore, and Steiner [ref. 3] describe the use of SysML to analyze stakeholder needs; characterize the as-is system and enterprise; perform causal analysis; specify mission requirements; capture measures of effectiveness; define the scope of the to-be system; and analyze system requirements in terms of scenarios, functions, critical properties, and constraints. These form a solid foundation so that the subsequent architecting process can be rigorously related to the customer's intent.

B-3.1.3 Flesh Out Boundary Information

There is a certain kind of operational information that is helpful in fleshing out the expectations on the system: who is going to use the system; temporal aspects of activity expected of the system; or the process, policy, and organizational strategies that contribute to solution of the needs expressed. This kind of information is often expressed in scenarios, use cases, concepts of operation, or design reference missions.

There are two potential layers of operational information. The higher layer describes how the system interacts with the environment; a lower layer describes how the elements of the system interact with each other. At the conceptual stage, the higher layer is the important one. Delving into the lower layer too soon has the same kind of trap that white-box system boundaries do; it may obscure the work needed to understand relationships of the system with its environment.

Models such as sequence diagrams, functional flow block diagrams, or business activity or choreography diagrams are helpful for recording such information. They should be unified with the physical and functional models defining the system boundary.

The models can be tested against parties in the environment, by simulating system activity at the black-box level. As in the boundary comparisons mentioned earlier, the goal of black-box activity simulations is to reveal misunderstandings between the owners of the system-of-interest and owners of systems in the environment.

For software work, it is important that sufficient information regarding operational information that it is possible to explain how the system will interact with the environment from the software viewpoint.

Ingham *et al.* [ref. 12] report successful specification of system structure and behaviors using SysML, as well as automatic generation of the functional design descriptions. They found that using a system model provided a single, unified source of information and reduced the risk of inconsistency or divergence.

B-3.1.4 Find the System Functions

The black-box operational models of the external system interfaces should be extended to white-box descriptions of the internal activity needed to fulfill the external interface expectations. There are well-known strategies for accomplishing this, such as are described by Buede in reference 5, Chapter 7 (for system functions) and Chapter 10 (for interfaces), or by Friedenthal, Moore, and Steiner in reference 3, Sections 3.4, 8, 9, and 10. The resulting functional description of the system will consist of threads of internal activity that are derived from the desired external activity. These can be used to allocate functions to system components and their interfaces, which in turn drives the requirements on software interfaces. Having a functional design makes it possible to (a) explain what must be done by the physical system components in order to achieve something greater than their mere juxtaposition, (b) express what constitutes acceptable or unacceptable software interfaces, and (c) construct useful threads of verification and validation of software interfaces based on the larger system need.

Fluhr [ref. 18] has described a set of formal processes and procedures for developing functional models of a major flight mission, resulting in detailed operational models that can be used to automatically generate operations concepts, summary timelines, and detailed timelines.

B-3.1.5 Prepare for Downstream Engineering Activity

After the conceptual phase, the project engineering team will proceed to define software interfaces. That raises a question: does the team understand the conceptual spaces in which the software interfaces will be defined? If not, it will be difficult to proceed. In the cases where particular spaces are not well understood, some effort should be spent to define them *before* the definition effort begins.

As part of defining the conceptual spaces, it may be necessary to define terminology and conventions of expression to reduce the difficulty of communication and the chance of error or confusion. Jenkins and Rouquette [ref. 13] have developed substantial ontologies and modeling tools that cover business processes, systems engineering, and fundamental phenomena. These can be used to define some aspects of the conceptual spaces. However, there is not yet a specialized ontology for software fundamentals equivalent to what is available for the other domains.

The use of models to facilitate software interface definition is recommended. However, for that to work, the modeling environment must be able to distinguish the modeling spaces in which software interfaces exist. Without that, it will be difficult to address issues such as layering or illusions of separation. Some effort should be spent in this phase to prepare the modeling environment to be able to represent the various conceptual spaces involved. It may also be necessary to develop new well-formedness rules to enable model-based detection of misapplied concepts, beyond the usual ones of functional completeness, interface balance, and correctness of decomposition [ref. 42 (p. 41) and ref. 5, (Sidebar 7.2)]; or architectural information consistency, data completeness, and lack of ambiguity [ref. 43, Section 10.5]. This is a necessary precursor to

using model-based detection to correct misapplied concepts, so that early errors can be corrected before they harm the engineering effort.

It is possible, perhaps even likely, that the overall organization is currently using management and engineering processes for interfaces that revolve around the properties of the material world. For example, there may be a strict hierarchical decomposition that prevents common components from appearing in many places, there may be fictitious interfaces to top-level entities as described earlier, or the whole concept of functions and their interfaces may be excluded. These approaches will not work well for downstream software interface development. If this is the situation for any particular project, it is necessary to define new management and engineering processes that are rooted in the properties of whatever conceptual spaces are used to represent software interfaces. Indeed, it is necessary to define management and engineering processes that recognize, tolerate, and promote management of the nonmaterial world that software interfaces inhabit.

It is also important, if they do not already exist, to define management and engineering processes that recognize the existence of a top level software design, even in systems whose sub-elements are peers of the system of interest, or are legacy systems.

By the end of the concept studies phase, as part of the formulation agreement and the project plans, it should be clarified who has authority over the top-level software design and accountability for it. It will be difficult to decide later issues of partitioning and allocation of requirements to interfaces without that authority established. Also, as part of the formulation agreement and the project plans, the organization and work breakdown structure for software engineering should be defined, and accountability of element software engineers to the top-level software engineers should be arranged.

To support the formulation agreement, various project plans, and acquisition activities, it is necessary to determine which system functions, or more precisely, which parts of which system functions are to be accomplished by hardware and which by software. This should begin at a coarse level during the concept studies phase and be matured along with other system allocations.

During the concept studies phase, a project should be preparing a system for tracking technical performance, cost, schedule, and risk. In order to fulfill the expressed expectation from the systems engineering requirements to fully integrate software development with the systems engineering effort, software should be included in the tracking. The NASA software engineering requirements [ref. 24] require a number of software quantities to be tracked, and this should be integrated with the main project tracking systems.

A downstream activity will be the actual creation of artifacts that express software interfaces. Depending on a project's particular situation, it may be necessary to prepare a system for expressing software interfaces, both for interface requirements and interface design, based on the model. A lesson learned in context of the Constellation Program was to avoid fractionated efforts at modeling and conversion into artifacts, because of the engineering incompatibilities that arose.

Bayer *et al.* [ref. 14] reports successful application of models to several of the preparation challenges. They developed a system for tracking technical performance resulting in an equipment list and mass report produced directly from the model, as well as a power

consumption report. These were successfully subjected to completeness and correctness checks. They produced work breakdown structures from the model and achieved a high degree of uniformity in the analysis process that enabled them to run exactly the same analysis script on all three of their mission models. The simplicity of the script depended heavily on a lesson they learned regarding separation of the analysis from the model, which led them to adopt modeling patterns that aligned with the mission concepts instead of the analysis. Their team found that everyone needs to be trained to use the model-based approach successfully, but not to the same degree. A core team should be heavily trained and able to apply best practices in guiding the remainder of the team, most system engineers, including leadership, should be trained to develop models with help from the core team, and everyone should be trained to understand and express concepts using SysML standard notation.

Bayer also recommends tying expectations of any modeling efforts to project deliverables, not merely modeling solutions. This may need to be reinforced frequently, to keep the models focused on what actually has to be modeled to solve the problems at hand, rather than letting the models expand without restraint.

Somervill and Johnston [ref. 19] are currently experimenting with using risk-driven modeling, in which top risks are used to guide the areas of the model that should be most thoroughly developed. Areas noted elsewhere in this report as significant issues are good candidates for risk-driven modeling.

Oster [ref. 15] reports that the existing practice in modeling is dominated by focused activities within a particular discipline. The capability to support integration across discipline lines tends to be limited or missing, and existing integrations tend to be "point to point." The NDIA makes a similar observation in reference 30. Although software is just one among many other disciplines, successful interface development depends on the ability to share model information between software and the other disciplines. This suggests that it would be beneficial to the software interface effort to have an efficient capability for general information sharing between models in different disciplines. The reason this would be beneficial is that software interfaces are often connected to discipline-specific information such as equipment parameters or discipline-related functional threads. These connections are a source of manual intervention to share information between communities, with the attendant consistency problems that result from a manual approach.

Oster [ref. 15] also reports construction of an integrated set of models at Lockheed Martin that can be used across the entire product life cycle. The classes of models include the system concept of operations and requirements, system architecture (with models for test, cost, software, firmware or electrical, and system analysis), a three-dimensional CAD model (with models for mechanical analysis and producibility), a bill of materials, and an operations and sustainment model (with reliability and life-cycle cost models). These models are connected by an integrated data management layer that achieves integrations between major hubs of model instantiation. A structure similar to this, though with a more complete population of disciplines, is needed in the environment surrounding development of software interfaces.

Jenkins [ref. 16] has observed the general need for models to undergo transformations during the product life cycle. A model transformation is an operation that operates on a model and generally produces another model. Jenkins recommends preparing for automated transformations of various types, using conventional programming, specialized transformation

languages, or tool-specific integration or export or import tools. This area of effort is relatively new and would benefit from development of solutions to integrate system modeling with general computational services (MATLAB®, Mathematica®, Maple®), high-end simulation or analysis tools (Modelica®, Phoenix® ModelCenter, ARENA®), and specialized correctness checkers or provers (SPIN, Java™ Pathfinder).

Friedenthal [ref. 17] reports that tools in general are evolving towards an MBE paradigm and progressing toward greater tool-to-tool interoperability. However, he identifies a critical challenge in that model management will have to address configuration and change control of models, tools, and simulation data throughout the life cycle and will have to address security and access control (of the same entities). The solutions may include integrating system models into the product life-cycle management environment and model interoperability based on standards. These are new developments that would be needed and would have to be addressed during the preparation stages of a project.

B-3.2 Concept and Technology Development

B-3.2.1 Define Software Requirements on Systems outside Project Control

Systems may be outside project control because they are part of the environment, because they are legacy systems to be incorporated into the system-of-interest but are no longer in development, or because they are a new or modified system to be incorporated into the system of interest, but authority over them belongs to some other project (characteristic of an SoS). In the first and third cases, models can be used to clarify the expectations that the developers of the system of interest have upon the peer system. The system-of-interest's engineers can make a model of how they think the peer system works and ask the question "does it work like this," or the peer system's engineers can supply a model and make the statement "it does work like this." In both the first and third cases, agreement can be reached based on the model, after which both sides of the interface can incorporate the model into their own, locally governed requirements.

In the second case, all is the same except there is no agreement needed; the interfaces are what they are. The system-of-interest's engineers can incorporate the model of the legacy system into their own requirements and design the new system to match.

B-3.2.2 Define the Conceptual Temporal Software Interface Characteristics

At the conceptual phase of development, an initial decomposition of the system will be created. It is then necessary to decompose the conceptual functions to a level that can be allocated to the sub-elements.

The decomposed functions will eventually need to have their temporal characteristics defined. At the conceptual studies phase, this should be done on a conceptual level, defining major characteristics, such as which entity initiates or controls behavior of the interface.

For software work, it is important that sufficient information regarding the temporal interface characteristics be obtained so that it is possible to discern what is expected from the sub-elements from the software functional viewpoint.

B-3.2.3 Manage Software Interface Requirements and Constraints of Many Types

It was previously pointed out that there are a great number of possibilities for the conceptual spaces in which software interfaces exist. Additionally, software interfaces often exist in multiple conceptual spaces simultaneously. Hence, requirements on any particular software

interface may include requirements relevant to multiple conceptual spaces inhabited by the interface. These must be recognized and managed along with other requirements.

B-3.2.4 Plan to Overcome Software Interface Technological Limitations

Each technology for realizing software interfaces has some set of limitations: completeness for messaging architectures, conflicts of control for shared-memory architectures, loading of shared resources for networked and brokered architectures, etc. Plans should be made for the preliminary design phase to develop models that contain sufficient information to manage these limitations.

B-3.2.5 Select External Software Interface Architectures

The selection of external software interface architectures should be made in the conceptual phase because their characteristics are needed to guide decisions about partitioning and interface definition interior to the system of interest. Options are message passing, shared memory, network (pipeline, bus, start, ring, or mesh).

B-3.2.6 Find the System Functions Again

After the initial concepts of functions are defined, it becomes possible to discern relationships between them that reveal the need for new, derived constraints or functions. For example, defining the functions of attitude control and power generation may reveal that the power generation function needs attitude control to point solar panels at the sun or to keep radiators in the shade. Or, it may become apparent that a new function is needed to manage the state of battery charge in the spacecraft. Such discoveries will create new information about needed software interfaces. Therefore it is necessary to repeat the step of finding system functions, this time including the new information that resulted from software requirements and external interface architectures.

B-3.2.7 Partition the System and Software

Partitioning of the system and its software can result in software interfaces that are more or less likely to be successful. Considerations are update rate of information, complexity resulting from coupling of information between systems or software elements, robustness to changes in external conditions (including potential future changes of interface architecture), sensitivity to and management of faults, and ability to realize the desired emergent characteristics of the system as a whole.

Partitioning of the system and software faces the same issue of defining the conceptual spaces in which the software exists as did defining the system boundary. The issue should be handled the same way: express the interfaces in multiple conceptual spaces, define the differences between the physical and software boundaries, and check for and correct pathologies.

For the concept of system software interfaces to work, the assumptions of boundary and interior regions must be realized. Therefore, the software partitioning should create such regions, in particular, interior regions must be created where changes in the environment do not require changes in the software design, and vice versa.

Given that software interfaces may exist in multiple conceptual spaces simultaneously, it is necessary to check for illusions of separation, or equivalently, check for sneak paths between conceptual spaces that create unintended interactions between elements of the software

partitions. Depending on the project's situation, it may be necessary to develop well-formedness rules to enable model-based detection of illusions of separation.

B-3.2.8 Explore Emergent Software Characteristics

A search should be made for derived functions that arise out of the expectations. These can be traced using a requirements model, to ensure that the derived functions are engineered into the functions of the system. This is a necessary foundation for eventually allocating functions to software.

B-3.2.9 Define Models of Software Interfaces

Models of interfaces can be used to convey requirements. The technology exists today to derive implementations of interfaces from models [ref. 30, pp. 24–25], eliminating errors that can arise from translation into and out of documents containing natural language.

Operable models of interfaces can also be constructed early and exchanged between the software developers so that they are able to confirm compatibility between software items locally.

B-3.2.10 Define Responsibility for Software Interface Definition

Responsibility for software interface definition is straightforward in top-down design that involves the creation of new sub-elements; the software engineer for the higher-level system is responsible.

In the case of peer-owned systems, however, it is not clear which of the peers is responsible. A mechanism is needed to clarify the situation and to allow the peers to be synchronized in their understanding of the interface. A single model of the interface, with shared governance of that model, can make this easier.

In one solution, in an interface shared across a NASA and contractor boundary, the development teams on both sides agreed to co-own the upper level model of the interface. The development of this upper level model was a shared responsibility, and the development team was composed of members from both NASA and the contractor. In this way, legal and intellectual issues were mitigated using a single interface model shared across the boundary. The interface verification and validation was developed from the shared interface model.

B-3.2.11 Describe Software Interfaces from the Models

Models of interfaces can be used to describe interfaces. The technology exists today to derive some aspects of interface descriptions from the software that implements the interfaces [refs. 12 (p. 10) and 44 (pp. 18, 37)]. This can eliminate errors that can arise from translation into and out of documents containing natural language.

The language for expression can be UML, SysML, or BPMN. IDEF 0 may be a possible alternative, though it may be limited in its ability to express multiple layers that are arranged in anything except a strict hierarchy of nonoverlapping components. More on this topic is discussed in Section B-4.2.

B-3.2.12 Plan Software Interface Content

Software interface content must address all the relevant aspects of the conceptual spaces in which a particular interface exists. The relevant content may not be the same for all interfaces, and a one-size-fits-all content standard may tend to obscure relevant information. Without

intending to be complete, the range of content that might be included for software interface descriptions is illustrated in Attachment B.

Consideration should be given to using a selectable content standard based on the models that define the conceptual spaces in which an interface exists. Plans should be made for defining the space-specific details during the preliminary design and technology completion phase.

Plans should be made for reference models for protocols or commonality standards to be used later in detailed software interface specification.

B-3.2.13 Track Software Interface Characteristics

Models can be used to aggregate software interface characteristics across the multiple spaces in which the interfaces are defined. The results of these models with respect to software interface characteristics should be incorporated into the project's tracking systems.

Friedenthal, Moore, and Steiner [ref. 3] explain in detail the use of SysML to express relationships between interfaces, especially using the "flow" construct. Illusions of separation can be checked using the "uses" and "realization dependencies" constructs and multiple model views corresponding to the layers of interface interaction. Requirements satisfaction can be tracked using SysML requirements diagrams and associated tracing.

B-3.2.14 Plan Model Exchanges

Plans should be made for the ongoing exchange of models for requirements or description purposes. This may involve tool-to-tool or language-to-language translations. This also may involve upgrades of the information in existing models either to bring content up to date as designs mature, or to add information needed to use legacy models in a new system.

B-3.3 Preliminary Design and Technology Completion

B-3.3.1 Define the System Context Model

Partitioning criteria previously mentioned in the context of interface pathologies can be used to allocate functionality, persistent data, and control information among the functional and physical components of the system. System interfaces should be considered logical components that are then allocated to hardware, software, or other types of components.

Context models and discipline-specific or "domain" models describe things beyond the scope of software implementation. That is, they describe the problem domain, not the solution (software) domain. Context models define the physical parts of the system—external systems, users, interfaces, communication channels, etc. Domain models, of which software is one, describe the informational context of the system—which artifacts exist, are produced, and are inputs or outputs, and how these elements relate to each other. An important note related to interfaces is that all of the data in these models will exist irrespective of what software solution is developed. If the data model changes, then the understanding of the problem must change.

A good practice is to avoid writing implementation choices in the initial use cases. This can be accomplished by limiting the concepts written in the use case descriptions to only those defined on the context or domain models. The reason for this practice is that there is a risk of becoming unable to evaluate alternate solutions to the system problem if the problem itself is expressed as a solution. This also relates back to the black-box or white-box issue discussed earlier; discussion of internal details can obscure the lack of information about required external characteristics.

B-3.3.2 Develop System Functional Architecture

Functional modeling, also called component modeling, focuses on building the functional architecture of the system by breaking the problem domain into a set of nonoverlapping and collaborating functional components. Logical decomposition of the system into logical components includes external interface components, application components, and infrastructure components.

The concept of a nonoverlapping decomposition should be taken literally at first, but later relaxed as commonly recurring patterns of function appear (analogous to "spray" or "bubbles" in Figure B-1-2). A check should be made for such patterns because they form the basis of common solutions deployed at many locations in the system and resulting system efficiencies. Communication, control, and data access are frequent examples of these. Once common solutions are identified, the nonoverlapping decomposition should be refactored to allow the use of commonality; this may result in a different decomposition, and it may include overlaps where common components are either (a) grouped into a centralized solution, or (b) recognized as distinct, nonoverlapping instances of a single type of component or a family of related types of components sharing a common design.

Modeling should be done at least at two different levels of granularity: the analysis level and the design level. The main difference between the two levels is the degree of specificity that they illustrate. The design models build upon the analysis models by adding details that contain enough information to facilitate the third level of modeling: the UML implementation model, where the physical structure of software components (such as executables) are defined, along with their run-time deployment onto system hardware. Analysis and design models can be called a macro design (the kind of thing that is done during concept and preliminary design phases), while implementation modeling is a micro design (the kind of thing that is done during critical design phases). A check should be made to see whether the implementation model is created before the analysis and design models; as with black-box or white-box models, a premature discussion of the implementation model can obscure the lack of knowledge regarding the desired functions.

Component diagrams can be used to model the logical architecture of a system. Sequence diagrams can be used to model not just software components but also system level behaviors, that is, how particular system components should interact.

B-3.3.3 Design System Physical Architecture

Logical components should be allocated to physical components (even when allocated first to software components). Allocation decisions help answer performance, reliability, security, and other system issues that impact interfaces.

Deployment diagrams should be used to model the physical architecture of a system by first showing nodes and devices and connecting them using communication paths. Next, the components that run in each of the physical nodes are modeled, followed by the applications that run on the different nodes and the components that make up the applications (for example, application artifacts may be wired to nodes through the 'deploy' connector and wired to components through the 'manifest' connector). In order to show the way in which the system interacts with external applications, artifacts can be used to represent the external application.

A deployable software node, executable, or module should be considered to be a physical component in this context.

B-3.3.4 Find the Last of the System Interface-related Functions and Characteristics

The process to design emergent software interface characteristics begins by identifying repeated patterns of interactions (interfaces are essentially drawn from interactions with an outside system or environment). These patterns become a key element of the architecture of the interfaces. Interaction use cases can eventually be defined in the sequence diagrams and formally tracked in models. Software system functions such as shut down, start-up and fault management functions (FDIR) should be derived and modeled. Many of the performance characteristics of these functions affect or are affected by the data provided across interfaces. The inputs and outputs of scenarios, use cases, and system state machines associated with scenarios and system behaviors are a good source of interface information.

Interface characteristics include:

1. Priority assigned to the interface by the interfacing entities
2. Type of interface (e.g., real-time data transfer, storage and retrieval of data, etc.)
3. Characteristics of the individual data elements that are provided across the interface, such as name, data type, size, format, unit of measure, rage, accuracy, priority, timing, frequency, source of data, and constraints (security, privacy etc.)
4. Characteristics of the data aggregates (messages files, records, etc.) provided across the interfaces, such as name formats, data structure, sorting or access characteristics, priority, timing, frequency, volume, sequencing, constraints, etc.
5. Characteristics of the communication method used by the interfacing entities, such as name of the communication link, message formatting, flow control, data transfer rate, routing, addressing, naming conventions, encrypting, authentication, whether the flow is continuous or discrete, etc.
6. Characteristics of the protocol layer, such as name, priority, packetizing, error control and recovery, synchronization, connection information, reporting features, etc.

These characteristics should be tracked carefully because they are important for compatibility checking.

Also, there are multiple sources for software interface characteristics and constraints. These sources include system decomposition (especially the interface allocations that result), interface standards and protocols, and constraints from other systems.

A good practice in defining interface functions and characteristics is the use of the concept of interaction boundary, which incorporates formal gates where information that leaves or enters the boundary must be formally defined. Modeling tools and techniques are particularly adept at realizing the design of these boundaries.

B-3.3.5 Propagate System Flows to System Boundaries

Interactions may be nested and further decomposed. It is not sufficient to identify the inputs and outputs from and to the nearest components to the system interface. Data must be flowed through the entire system from the inception of the data to its final disposition. This is another justification for using functional and behavioral modeling (in addition to structural modeling) to validate the SoS design at the interfaces; without such modeling, the large scale, long-range flows and interactions are not apparent.

Flows identified by internal block interfaces should be propagated through the system until they reach the system boundaries. There, they should be then captured as formal software interfaces between systems. Interactions include the interactions between software and system interfaces, hardware interfaces and other boundaries.

SysML can use ports to represent an interaction point at a boundary. Flow ports describe flow in and through a boundary point. Service ports describe what services are required or provided at the boundary.

Software models should use flow ports to represent data flow and performance behaviors across interfaces. Flows may be continuous or discrete.

Often, the process of elaborating the high-level design leads to the discovery of new interface needs. Typical interface design issues include incompleteness, overdetermination, interface

incompatibility, transection across data complexity (requiring knowing too much information at the interface), transection across excessive data rates, or any of the other interface pathologies mentioned in Section B-3.1.1.4.

In summary, when defining software interfaces, look for other-than-system boundaries, for example, hardware boundaries, software layers, interfaces to protocols and legacy or COTS software, etc., to find the true data sources and sinks.

B-3.3.6 Describe Software Data Flows

Data flow modeling is the process of identifying, modeling, and documenting how data move around an information system. Data flow modeling examines processes (activities that transform data from one form to another), data stores (the holding areas for data), external entities (what sends data into a system or receives data from a system, and data flows (routes by which data can flow).

SysML "items" should be used to model software flows. SysML items may be blocks, value types, or signals. They are good for the description of physical flows, as well as information flows. An item that flows may be a complex structure like a 1553 interface. Wait time and other temporal considerations use signals.

Service ports should be the common mechanism to model behavior across software components. Model Items may be defined in a library containing "standard" interface items and or special ones. These libraries of interfaces provide a crucial aid for generating interface documents for a system using special document generation add-ons to modeling tools.

B-3.3.7 Define Software Interface Standards

Whenever possible, a project team should consider using an industry-recognized standard for system interfaces.

Conventions and standards should be required to ensure consistent representation and styles across modeling activity. Interface standards may include operational characteristics and acceptable levels of performance.

Interface standards and conventions must be modeled and reviewed to highlight gaps or inconsistencies in the planned use of the standards in the SoS.

For complex systems, it is a recommended practice is to use formal interface taxonomy to specify physical and logical classification of interfaces.

SysML has many constructs that support the incorporation of standards into the interface design of an SoS, such as blocks, symbols, and user-defined stereotypes.

One notable caution: tool-specific notations may not be easily compatible with standards.

B-3.3.8 Select Internal Software Interface Architectures

A unique feature of software interfaces is the presence of cooperating entities in separate systems. Data and information may be produced in one system and carried across the interface to the other system, where it is used by a low-level software component that is far removed from the interface logical boundaries. For successful verification and validation of the interfaces between these cooperating entities, a model of the behavior of these two systems must be developed.

Checks should be made at this stage to confirm compatibility between each pair of ports involved in the interfaces, according to the characteristics identified in Section B-3.3.4.

B-3.3.8.1 Integration Brokers at the Software Interfaces

Integration brokers provide a mechanism for communicating with the outside world using common XML files. Communication can take place between different program applications or between one part of the system to other systems or third-party systems. To subscribe to data, third-party applications can accept and process XML messages posted by the sending application using the available connectors or by adding a custom-built connector to the integration gateway. This topic primarily covers publishing outbound asynchronous messages. Once an integration broker is selected, it may be necessary to select the implied interface methods to or from it that are associated with that selection.

The Common Object Request Broker Architecture has the ability to supply definition objects from one application to another, expressed in OMG® IDL [ref. 45].

There are tools that verify the compatibility of the interfaces of standard ports by verifying the compatibility of the operations on the interfaces. This may be too restrictive, so SysML has left the issue of compatibility undefined or more open. However, there is a well-developed formalism for detecting mismatches described by Gacek [ref. 9].

In complex systems, interface software is often built with and interfaces with COTS code that is normally provided for communication protocols.

It is also important to note that software typically interfaces at locations other than physical connections, that is, software interfaces are "produced" inside a module deep inside the flight software design and far from the physical interfaces. Similarly, software modules on the other side of the interface "consume" the software information coming through the interface. These consuming modules also exist far away from the physical interfaces. Some effort may be needed to discover these. For example, a client server interface may reside on the user's machine or on an FTP server inside a system. Or, a Web interface may need a Web browser on the user's machine.

B-3.3.9 Identify Layers of Compatibility

At the interface level, compatibility analysis should be performed based on the component specification on either side of the interface and must consider three distinct aspects: structural, behavioral, and functional. The structural aspect concerns the static features of a component and corresponds to the set of required and provided operation signatures of the component. The behavioral aspect defines constraints in the invocation order of provided and required operations. The functional aspect describes what the component does, not necessarily going into details of its implementation. Both the behavioral and functional aspects will be affected by performance requirements, which may be a factor in compatibility.

When software or models on either side of the interface are not designed for compatibility (which is usually the case when two different systems are developed independently but must cooperate operationally), the lack of standardization in the specification makes the analysis of compatibility across a system interface difficult.

UML and SysML provide mechanisms to deal with components but do not establish a standard for complete specifications. Thus, each component or side of the interface has its own state

machine (SM) representing its externally observable behavior—the sequence of required and provided operations performed during the component's operation. This is the crux of the compatibility analysis problem.

At the model level, compatibility refers to the ability to connect ports without violating constraints. There are default compatibilities between ports based on name, type, and direction mapping, and we can conjugate ports to reverse direction of flow properties.

Connectors typed by associations can assert compatibility in several ways: a) among different port types, b) to specify protocol interactions via owned behavior of association, and c) to constrain end types in support of engineering analysis.

Interface compatibility analysis must be performed during the design phase of the life cycle. A lesson learned is that if the interface compatibility analysis is deferred until the implementation phase, the cost of correcting incompatibility errors will skyrocket.

B-3.3.10 Allocate Functionality to Software or Subsystems on Either Side of the Interfaces

Software functions are allocated to software components on either side the interface. The interface itself may include the capability alter the contents and/or the behavior of these functions. For example, the interface may contain commands to add or delete, initiate or cancel a function. The interface may transfer input data and parameters from one part of the system in order to affect the behavior of a function in another part of the system, or it may carry the output/results of executing a function in one subsystem to recipients across the system.

From a modeling perspective, one needs to define the correct modeling behavior of the allocated functions, as well as the interfaces that affect or affected by these functions.

The interface model must not be limited to the functional interfaces with the nearest components to the system interface. Interface modeling data must be flowed through the entire hierarchical functional design of the system from inception to final dispositions. Propagating the functional interface modeling to all of the functional design layers of the system will enable the use of functional and behavioral models to validate the SOS design at the interfaces.

B-3.3.11 Allocate Data and Control to Software Interfaces

It is a good practice to explicitly allocate data and control to software interfaces. Allocation to system interfaces follows, and these are a subset of the total software interface allocations and will need to be identified as allocated to or from system boundaries.

The interface characteristics previously identified should be reflected in the model, such as data types, size, formats, units of measure, range, value enumeration, accuracy, frequency, security characteristics, data sources and sinks, and other constraints.

Check functional completeness as a first step and then do interface balancing on the functions. After that, do end-to-end interface balancing to ensure input and output balance, and run through scenarios to see if the system works as intended.

B-3.3.12 Derive Software Interface Designs to Meet System Emergent Characteristics

Emergent system characteristics may not result in any new hardware interfaces, but likely require new aggregation or analysis of the interface and software data.

In order to accommodate emergent system characteristics, interfaces must carry adequate capacity and performance margins and have behaviors that are compatible with the desired emergent characteristics.

When interfaces are traced in different conceptual spaces or layers, unintended interactions can be found between items that seem separate at another level. For example, tracing command data exchange messages (DEMs) between two systems allows for design verification of requirements regarding sending and receiving these command messages. However, if one needs to verify that every sent command has been received by the intended recipient, one must first understand whether the system allows packing of one or more commands inside each command DEM. Tracking DEMs sent to and received by a system hides the added complexity of needing to track the next layer of detail, which are the commands themselves, if one desires to build command robustness into the system.

B-3.3.13 Derive Software Interface Designs to Meet Interface Requirements and Constraints

There should be explicit interface requirements on software (e.g. exchanges of commands, files, status parameters, and telemetry). Additional interface designs should be influenced by hardware connections, communication standards, and network protocols.

Consider again the issue of unintended interactions discussed in the previous section. These may arise because of nesting of flows. For example, if one or more system commands may be embedded in a single command message, then one needs to elaborate the constraints and the requirements associated with command handling (such as what to do with a missed, garbled, or invalid command). These are instances of appropriate requirements at the command processing level.

SysML enables robust interface modeling capability and the ability to query the traceability between design and requirement models (up, down, and sideways) for identifying structural, logical, and behavioral gaps and inconsistencies.

SysML capabilities, now extended to include nested ports and flows, allow the modeling of diverse interfaces and the ability to specify port compatibility.

B-3.3.14 Aggregate Software Sources and Sinks

Software modules aggregate internal sources and sinks into complete flows. At the SoS level, this aggregation needs to extend beyond the software within an individual system, across the interface and reaching to the software sources and sinks in the other interfacing systems. This aggregation is needed in the context of defining data architecture for the SoS.

B-3.3.15 Define Software Interfaces between Models

Models developed independently and using different tools may need to exchange information with other models in order to operate, or information may need to be exchanged between models to support integrated analysis of system interfaces.

SysML interfaces to engineering analysis tools are different from the interfaces with requirement analysis tools or interfaces with verification tools. This raises the need to ensure model-to-model interfacing as a separate concern from the system-to-system interfacing problem to be analyzed with the models. Once modeling interfaces are understood, modeling exchanges should be enabled and system interface analysis can proceed.

B-3.3.16 Detect Software Interface (In)Compatibility

System interface modeling identifies the characteristics and attributes of the interfacing elements, at many different layers of the interface communication stack [ref. 46]. One can model the characteristics of the individual data elements (such as data items or parameters), the data aggregates (such as files or messages), communication methods, and protocols in addition to the physical modeling.

Compatibility may be analyzed at each of the software layers of the interface. These attributes must be well understood on both sides of the interfaces in order to detect incompatibilities. Incomaptibility could mean mismatched types, missed specifications on either side of the interface, missed directions, or missed software interfaces altogether. Methods for detecting incompatibility have been described by Gacek [ref. 9] and Yakimovich, Travassos, and Basili [ref. 8].

B-3.3.17 Discover Preliminary Unintended Consequences

The process of designing complex systems with complex interface always runs the risk of creating unintended design features or characteristics of the system. The major contributors to these unintended system consequences are the following:

1. Complex relationships among design entities: Difficulties in solving problems often stem from the fact that problems do not occur in isolation, but in relation to each other.
2. Feedback cycles: Issues relate to each other in specific ways based on feedback cycles. It is difficult to think of all the system actions and reactions in the design process.
3. Latency: Feedback cycles involve delays, which make system and interface behaviors harder to predict.

Modeling solutions can help on all of these fronts. Key system and interface relationships, feedback cycles, and latency characteristics can all be modeled in the level of detail needed to better understand and observe the system behavior and verify that it meets its intended functionality.

B-3.3.18 Iterate the SoS Software Design

Analyzing the various views of the modeled interfaces can assist in the design process. Physical views and functional views can be queried in order to answer questions about the completeness and the static characteristics of the interface design. This technical insight supports the optimization of the interface design, which is a key objective of this phase in the life cycle.

Executable models and the ability to run themes and variations on key interface scenarios using models allow a deeper understanding of the behavioral characteristics of the interface. They may answer questions about the key performance and timing bottlenecks in the interface and, hence, contribute to a better understanding of the system behaviors and overall performance during critical events, before a design is locked in.

Design analysis cycles, supported with static and/or executable models, may be used to iterate and refine the software design.

B-3.3.19 Verify Preliminary Software Design

As mentioned earlier, executable behavioral models can verify many aspects of the end-to-end software design. Descriptive models are not sufficient for software verification because they lack the ability to verify timing and performance aspects of the system. Executing mission timelines with models enables the verification of proper system behavior at critical mission phases.

Appendix G of the NASA Systems Engineering Requirements [ref. 23] gives the entrance and success criteria for a successful PDR, which include interface criteria. Many of the answers to interface questions at PDRs may be provided using models and simulation techniques. Additionally, milestone products and documents may be generated directly from the models.

Models allow more complete verification of key interfaces and requirements, while traditional PDR review methods allow (at best) for spot checks of these interfaces and requirements.

B-3.3.20 Plan Software Interface Verification and Validation

The verification of software interfaces involves the demonstration that the interfaces meet the design specifications.

Planning for interface verification includes the selection, inspection, testing and demonstration of interface data structures, functions and behaviors. Planning also covers the support tools and the test and simulation environments used in verification. Finally, planning also includes the management of resources, schedule, constraints, dependencies and risks of model-based validation and verification activities. The methods of verification include inspections, peer reviews, audits, walkthroughs, analyses, simulations, tests, and demonstrations.

The primary method through which model-based design achieves verification and validation is through testing and simulation because iterating in a modeling environment is fast and easy. Many organizations do some form of modeling and apply simulation in an ad-hoc manner that does not maximize the potential verification benefits. Limited modeling of individual components can still be useful to the verification process of interface design, as it may identify gaps in the interface requirements or specifications. However, to get the full benefits of model-based techniques, it may be necessary to model the whole system (i.e., all of the relevant interface functionality and behaviors). Behavioral models can run and re-run time critical activities under a variety of initial conditions and test constraints, long before the availability of hardware-intensive test environments. Early functional and behavioral modeling of interfaces enables the early identification and correction of system and software design defects.

The case for planning model-based validation activities is similar to the case made for verification. The goal of validation is to demonstrate that the "right software interface products" have been built. So the planning emphasis of the MBE work should be to enable functional and behavioral analysis of the entire system, in a single environment. Therefore, the validation process relies heavily on modeling as much of the system behavioral attributes as possible.

B-3.3.21 Design Software Verification and Validation System

System modeling and simulation tools, such as Simulink, help streamline the task of designing and verifying complex algorithms without expensive hardware. In place of hand-coded, hard-to-maintain simulations, designers can quickly develop complex algorithms, system models, and modeling environments to test their algorithms before committing them to hardware. Model-

based approaches to system design include automatic code generation, used first to support real-time testing in prototyping systems and later for deployable embedded code. Today, model-based design is used in a variety of applications and functions, including controls, data and image processing, audio, communications, and signal processing. The interfaces to these functions and applications could clearly benefit from model-based verification approaches.

Developing tests in parallel to design and development allows early detection of potential problems, and it significantly reduces the cost and time required for fixing them. By thinking about testing while you develop the model, you will design better for testability, thus ensuring that the design is fully testable.

One of the primary benefits of model-based design is the opportunity to do rigorous verification and validation in parallel with all other development steps, especially early in the development process. You can maximize these benefits using a series of best practices that adopters of model-based design have discovered through hard experience. Practices such as developing tests along with models and reusing model tests on code and hardware, among others, can significantly reduce the risk of a development project missing quality or delivery goals due to the discovery of errors late in the process.

Model checking tools and other automated verification and validation systems should be used to check life-cycle activities: requirements, design, and implementation. Also, it is likely that bridge products or intermediate formats may be required to ensure success of the validation and verification activities. Off-the-shelf validation and verification tools and applications, linked together with bridge products, allow for integrated functional and behavioral analysis of the system. This ability to verify and validate the broad system functionality is critical to the early detection of design incompatibilities and for verifying the end-to-end system capabilities and behaviors.

Friedenthal, Moore, and Steiner [ref. 3] describe a "flow" construct that can be used to represent relationships between interfaces. The flows can be used as the basis of verification and validation of the software interfaces, providing a convenient way to tie them to higher level system requirements.

Ingham *et al.* [ref. 12] have demonstrated model-based tracking of associations between functions, physical architecture (including electrical system modeling), and behavior for the purpose of verification, including automatic generation of functional design documents and test scenarios (which relied on simple substitutions of test facilities for system actors).

Vera *et al.* [ref. 20)] currently have a capability of creating an up-to-date "integrated functional analysis report" and a traceability report for design verification. In one case, the users found an initial 80-percent discrepancy rate using this capability, which motivated a rapid series of design changes that reduced the discrepancy rate to a low level.

Oster [ref. 15] reports that Lockheed Martin has the capability to maintain an interface baseline using SysML internal block diagrams and "message" model elements. These document the hardware and software interfaces consistently and tie system behaviors to the interfaces. The tie between system behavior and internal interfaces can be used to structure verification activities around system behaviors.

B-3.4 Final Design and Coding

B-3.4.1 Design Detailed Software Interfaces

Given a software interface architecture, descriptions of software flows and behaviors, and software interface standards all selected during preliminary design, the detailed software interface design will focus on enumeration of data, refinement of interface behaviors, and selection of specific parameters of the interface architecture. These can be captured as refinements to the interface models. The resulting detailed interface models can be exercised to verify compatibility between the interfacing elements.

B-3.4.2 Manage Changes

Software change management can be automated to some degree if the dependencies on model characteristics are captured in a machine-readable form and an automated workflow system is in place. Dependencies can be identified *a priori* from the software architecture. Also, dependencies can be discovered over time from a record of access to model information. Parties who access model information are likely to have a concern if it changes; this can be confirmed in an automated dialog with the accessing party.

B-3.4.3 Discover Refined Unintended Consequences

Collections of detailed interface models can be exercised in the context of simulated missions to confirm that there are no unintended consequences to the choices made during detailed software design or that appear unexpectedly as a result of earlier software architecture choices.

B-3.4.4 Define of Software Manufacturing Process

The software manufacturing process can be captured in a model. The model can be used to automate workflow and to audit compliance and completeness of the process as executed.

B-3.4.5 Determine Readiness for Software Coding

The completeness and consistency of the requirements can be checked automatically by examining the models that convey the requirements. Incompletenesses and inconsistencies can be justifications for postponing the final software coding or for identifying risks associated with partial codings. Standards for readiness can be expressed in a model and can be automatically evaluated from the identified issues. This can improve the quality of decision-making regarding progress on software development.

B-3.4.6 Code the Software

Some code can be generated automatically from structure, data, and behavioral models if these are available from interface requirements. This greatly decreases the introduction of errors resulting from misinterpretation of natural language. Documentation of the as-built interface can be generated directly from code.

Oster [ref. 15] reports using scripts to automatically generate interface software directly from a system model that includes SysML internal block diagrams and "message" model elements.

Ingham *et al.* [ref. 12] reports integrating a spacecraft's electrical system model with the behavior model and the initial generation of the electrical interface function lists. From there, it should be possible to generate detailed hardware-software interface descriptions and associated code using scripts from the model, analogous to what Oster reports in reference 15.

B-3.4.7 Discipline to Software Interface Definitions

Discipline to software interface definitions can be enhanced by models in two ways. In the first way, if code is generated automatically from models and detailed descriptions are generated automatically from code, discipline to the definition arises automatically. In the second way, if code or detailed descriptions are developed manually, the actual interface can be verified against the model in the interface requirements or documentation of the as-built interface. This will produce evidence of deviations that can then be used to justify corrections.

The software coding does not have to be complete for discipline based on models to be accomplished; it can be done when stub software routines are defined to the extent needed to mimic the interface.

B-3.4.8 Implement the Software Verification and Validation System

The verification and validation process can be captured in a model. The model can be used to automate workflow and to audit compliance and completeness of the process as executed. A large part (but not all) software verification and validation will concern software interface structure, content, and behavior.

Implementation of the software verification and validation system should begin during the software interface design process. This can be done automatically to some degree based on the models of the software interfaces and architecture. Some must be done as real fabrication to have actual hardware on which to verify or validate the software.

B-3.4.9 Apply Software Interface Standards

When there is a need to incorporate large amounts of detailed interface information from references, as can arise from protocols or commonality standards, the information can be expressed as reference models. Then every interfacing party can incorporate the reference models into their software development, thereby avoiding incompatibilities that arise from a multitude of locally developed models for the references. This approach also avoids the cost of developing a multitude of local models.

B-3.4.10 Verify Final Software Interface Design

The as-built software interfaces can be tested automatically against the model in the interface description. This can improve accuracy and completeness of verification, can eliminate confusion as to which party to the interface is at fault for incompatibilities, and can reduce the calendar time needed to accomplish verification compared to manual methods.

B-3.4.11 Verify Using Virtual Software Implementations

In larger or high-risk systems, it may be desirable to verify the understanding of the software interface before investing in software implementation. The software can be "virtually fabricated" by placing the interface models into a discrete-event simulation engine, which then can be operated to demonstrate the expected behavior of the software.

B-3.5 System Assembly

B-3.5.1 Validate Using Virtual Software Assembly

In larger or high-risk systems, it may be desirable to validate the overall software architecture prior to investing in software implementation. A virtual software implementation can be

accomplished by connecting the models of software interfaces from the requirements specifications to form an integrated behavior model of the entire system. This integrated model can be "virtually assembled" by placing it into a discrete-event simulation engine, which then can be operated to demonstrate the expected behavior of the entire system.

B-3.5.2 Assemble the Software Verification and Validation System

A model of the software verification and validation system should be made during design. Assembly can be done automatically to some degree based on the model. Some must be done as real fabrication to install and connect actual hardware elements.

Ingham *et al.* [ref. 12] are currently planning to document electrical integration procedure requirements from a spacecraft's system model. Further, they have automatically generated test procedures based on the required system behavior by replacing system actors with test personnel and facilities.

B-3.5.3 Test Conformance to Software Interface Design

The software interfaces can be tested automatically against the model in the interface description. This can improve accuracy and completeness of verification, can eliminate confusion as to which party to the interface is at fault for incompatibilities, and can reduce the calendar time needed to accomplish verification compared with manual methods.

The as-built software interfaces can be verified against the models in the interface requirements or documentation of the as-built interface in two ways: (1) by means of a model of the other system that interacts with the software under consideration, or (2) by means of a model of the system under consideration that is executed and its results compared with the actual software results. Both of these approaches will produce evidence of deviations that can then be used to justify corrections.

B-3.5.4 Determine Readiness for Integration and Test

The completeness and consistency of the coding can be checked automatically by comparing the models conveying the requirements with the models describing the as-built software interfaces. Incompleteness and inconsistencies can be justifications for postponing the integration or for identifying risks associated with partial integrations. Standards for readiness can be expressed in a model and can be automatically evaluated from the identified issues. This can improve the quality of decision-making regarding progress on software system integration.

B-3.6 Integration and Test

B-3.6.1 Validate Using Virtual Software Integration and Test

In larger or high-risk systems, it may be desirable to validate the overall software implementation prior to investing in software integration with the actual hardware. A "virtual software integration" can be accomplished by placing the actual software into a collection of virtual machines, which then can be operated to demonstrate the expected behavior of the entire system.

B-3.6.2 Integrate Software Components

The structure of the software system to be integrated can be expressed in the requirements and the as-built software model. The integration can be done automatically in some environments, based on the specified structure.

B-3.6.3 Integrate the Software Verification and Validation system

The structure of the software verification and validation system can be expressed in a model. Integration of the software verification and validation can be done automatically in some environments based on the model. Some verification and validation must be done as real fabrication to install and connect actual hardware elements.

B-3.6.4 Test and Verify Software Design

The software test and verification design can be generated largely automatically from the higher level model in the software system architecture. The test cases and expected results can be derived from the architecture. A nonautomatic part of the design is that human judgment is needed to assess which verifications should be done by test, analysis, inspection, or demonstration. However, the assessment process can be semi-automated with a workflow based on the software architecture model. This assessment can be forwarded automatically as part of the requirements on lower level elements, to notify their developers as to the verification evidence that will be expected. At the stage where the software to be tested has been built, the verification evidence available can be forwarded automatically to the verifiers.

Ingham *et al.* [ref. 12] have automatically generated test procedures based on the required system behavior by replacing system actors with test personnel and facilities. Friedenthal [ref. 17, p. 19] observes that model-derived test cases are a typical practice, while model-derived verification planning and test architecture are advanced practices.

B-3.6.5 Execute Software Tests and Verifications

Verification of software interfaces by test and demonstration can be accomplished largely automatically by supplying inputs based on the model and comparing outputs with the expectation from the model. In cases where human or hardware interaction is required, prompts can be automatically supplied to humans on what to do and what to expect in response.

Verification by analysis or inspection cannot be done automatically. However, the analysis and inspection requirements can be generated automatically from the system model and forwarded to the verifiers using a workflow system.

B-3.6.6 Capture Evidence

The test and verification design can specify in a model which evidence is required, where and when it is to be obtained, who is to obtain it, and who is to judge whether it is satisfactory. When the evidence is machine-readable, it can be sent automatically to its intended destination. Scheduling of tests and participants can be done automatically based on the model. When evidence can only be collected by witnesses, they can be supplied with prompts on what to do and what to expect in response.

B-3.6.7 Validate Software

Software validation can be accomplished by exercising the as-built software system according to design reference missions expressed in the model of stakeholder expectations.

B-3.6.8 Determine System Acceptability

System acceptance decisions can be assisted by reporting aggregated metrics specified by the system model and traced to metrics from lower level elements. These can be organized by the functional decomposition developed in the conceptual design phase, aggregating according to the

function tree, and rolling up results to support a single top-level decision as described by Breidenthal [ref. 35]. Aggregated metrics can also be used to assist in the computation of award fees.

B-3.6.9 Discover Final Unintended Consequences

The entire as-built system can be exercised in realistic operations to reveal any unintended consequences. The exercise can be automated in some cases based on the system model. This requires the availability of support systems (both machine and human) to stand in for the environment, which may not be possible in every case.

B-3.7 Launch

B-3.7.1 Configuring the Software System

The software system can be configured using the system model; build profiles can be set up based on system composition and interface connectivity specified in the system model. Software interface parameters can be set as defined in the system model. Where hand input is required, export configuration instructions directly from the system model.

B-3.7.2 Software Readiness for Launch

Models can be used to compute readiness metrics for software interfaces, including verification status, bug reporting, and software process status. Parametrics can be used in the system model to assess readiness of system functions, traced to verification evidence for components.

Breidenthal [ref. 35] points out that the single "yes" or "no" decision for software launch readiness can be decomposed according to the top-level functions, mission capabilities, and the timeline contained in a system model. Questions detailing criteria concerning readiness for flight can be developed and then evaluated using criteria for judging readiness of the system, suitability of the mission design, residual risk, etc. A practical plan for deciding readiness can be derived from the model.

B-3.7.3 Virtual Missions

Executable models can be used to simulate software interactions with real systems to conduct virtual missions. These can be used for early validation, training, and final mission readiness testing.

B-3.8 Operations and Sustainment

B-3.8.1 Training

Use executable models for training, similar to virtual missions but more focused to specific system-with-operator-in-the-loop interactions.

Use information in the model to develop training materials based on system structure, behavior, and interface characteristics.

B-3.8.2 Software System Diagnosis

Compare behavior of executable models to actual system behavior to detect either anomalous software operation or to adjust models to more accurately reflect actual system operation. When a hypothesis is formed regarding the reasons for anomalous operation, test the corrections with the executable model to mitigate risk of unintended consequences.

Use definition information in the model to look up the meaning of system control, monitoring, and performance data as it is observed in real time.

B-3.8.3 Acquisition of Software Data

Use models to compute performance metrics for software interfaces, such as loading and response time, compute margin, etc. Connect the model to real-time data acquisition so that derived quantities can be calculated relative to actual data.

B-3.8.4 Software System Upgrades

Compare behavior of executable models with and without upgraded software to discern the aggregated consequences of upgrading software on overall system performance. This mitigates the risk of unintended consequences from software upgrades.

Comparing the behavior is much more practical if, as a consequence of prior system modeling activity, there is a clear system boundary and carefully described system interfaces, such as can be obtained using the processes described by Friedenthal, Moore, and Steiner [ref. 3] for block definitions, interfaces, ports, and activities.

B-3.9 Closeout

B-3.9.1 Analysis of Software Data

Use models to compute higher level performance metrics for software interfaces, such as system effectiveness (e.g., monitoring of operator workloads, consumption of system consumables, and system failure history).

B-3.9.2 Software Reuse

Check and correct the final models to ensure that they correctly identify final software structure and interfaces. Although this step is not necessary for the project close out, it has tremendous institutional value in lowering the barriers to eventually inheriting the software and interfaces on later projects.

When the software is inherited, it is necessary to re-evaluate the software interfaces for sources of architectural incompatibility with the inheritor. Model checkers, if any, originally used to assess incompatibilities in the originating system can be reused to check for incompatibilities in the new system. This will entail merging the models of the originating and inheriting system and may entail adjusting the well-formedness rules to suit the inheriting system.

Capture final as-modified-during-operation software interfaces for inheritance on later projects.

Reuse of software is much more practical if, as a consequence of prior system modeling activity, there is a clear system boundary and carefully described system interfaces, such as can be obtained using the processes described by Friedenthal, Moore, and Steiner [ref. 3] for block definitions, interfaces, ports, and activities.

B-3.9.3 Software Disposal

Retain models, executable models, simulations as permanent records final software structure and interfaces for inheritance on later projects.

B-4.0 Methods and Practices

Software interface engineering relies on the methods and practices used to carry out the engineering. As such, it is critical that the methods and practices be effective at accomplishing the goals of systems engineering—developing an operable system capable of meeting requirements within opposed constraints—within the domain of software interfaces. Observations were made earlier about the need for management and engineering processes to recognize the unique or dominant features of software interfaces because they can attenuate or amplify the real difficulties inherent in software interfaces. The following provides more detail on what is needed to be effective.

B-4.1 Conditions for Successful SoS Software Integration

Maier (ref. 27) points out several important differences between systems-of-systems (SoS) and routine systems. The first of these is operational independence. That is, the components of a SoS have a purpose of their own and can perform their functions independently without interacting with the other systems making up the SoS. Secondly, the components of an SoS have managerial independence because their components are controlled by different governing authorities. Next, SoS are usually characterized by evolutionary development. They do not first appear fully formed; they evolve as functions are added, removed, and modified during the operation of the SoS. A final major characteristic is that SoS always have an emergent behavior; the behavior of the SoS as a whole is never embodied in any particular component. Rather, behavior emerges from the system as a whole. A minor historical characteristic of SoS mentioned by Maier, which has since been deemphasized, was that they have constituent parts that are geographically distributed.

Successful integration of software SoS, therefore, will depend on addressing the ramifications of these characteristics. The independent operational components will need coordination, as will the independent governing authorities. Moreover, since the component systems have an existence independent of the desired SoS, the coordination must be done in a way that is acceptable to the existing authorities. Mechanisms are needed to handle multiple concurrent evolutionary states, and emergent behaviors need specification, both desired and undesired.

Lane and Boehm [ref. 29] describe a number of engineering and management issues that must be handled by SoS lead system integrators.

For management, an SoS usually has a large number of stakeholders, development organizations, parallel and independent (or not so independent) developments, and decision "approvers." An SoS also usually has risks that cut across the component systems and cannot be managed solely within one system. As a result, numerous stakeholders' satisfaction (or dissatisfaction) must be managed. The development organizations and interacting developments must be managed, and it is not a hierarchy since the component systems have an existence of their own in some other collection of hierarchies to which they already belong. Because of the size of the number of parties involved, decision mechanisms need to be much more efficient than for single systems; decision mechanisms that are "just good enough" for single systems will not be good enough for the larger SoS. Also, provision must be made to mitigate cross-cutting risks, not just single-development risks. Lane and Boehm emphasize that integrated SoS solutions must be created using authoritative and accurate sources of information from the component systems. Obtaining authoritative and accurate information will usually require extra effort because of the disparate

regimes of authority, different tools, and sheer number of layers involved in conveying the information involved in integrating an SoS.

For engineering, an SoS integrator has a major issue to resolve: when to system engineer and when not to system engineer. The requirements are primarily at the SoS level and should only address the component systems with respect to their integration into the SoS. The SoS integrator should recognize that component system technical solutions, integration, verification, and validation activities are the responsibility of the component system "owners," lest the integrator's effort be distracted from the SoS goals and perhaps be overwhelmed by concerns of the component systems. The SoS technical solution, product integration, verification, and validation should focus primarily on the communications (or interactions) between the component systems.

As a contractual matter, the lead system integrator may or may not be responsible for actual development of component systems for the SoS. If they are, perceived conflicts of interest must be carefully managed. The goals of each component development must be carefully balanced to ensure the overall SoS success.

To summarize the ramifications for SoS engineering, requirements have to focus on the emergent characteristics, systems engineering at SoS level has to stay out of components, and the emphasis should be on interactions of components. That last issue alone tends to drive the engineering environment toward standard information exchange between models of the component systems. Also, suboptimization of components must be accepted to optimize SoS characteristics

Charette [ref. 28] compiled a number of characteristics of unsuccessful software projects, namely:

- Unrealistic or unarticulated project goals
- Inaccurate estimates of needed resources
- Badly defined system requirements
- Poor reporting of the project's status
- Unmanaged risks
- Poor communication among customers, developers, and users
- Use of immature technology
- Inability to handle the project's complexity
- Sloppy development practices
- Poor project management
- Stakeholder politics
- Commercial pressures

These negative characteristics should all be removed or converted to complementary positives to increase the chance of success for a software SoS. In addition to factors mentioned, Charette's list points out that unrealistic or unarticulated goals for emergent characteristics would inhibit success. From this, it is inferred that realistic and well-articulated goals for emergent characteristics must be in the specifications of the software SoS. Similarly, Charette's observations support the idea that estimation must be more accurate than for single system; that communication among customers, developers, and users must be more efficient than for single system; and that the ability to handle complexity must be greater than for a single system.

Jolly [ref. 1] present's a view that the relationship between software engineering and system engineering has changed fundamentally for developing space systems. His view is that software

and avionics have become the system. Earlier, it was pointed out there are conditions when this view is entirely accurate: either the system is in fact a software system or the complexity represented in the overarching system problem is primarily allocated to software. After a protracted study, Jolly's team found no "magic few" underlying root causes for difficulty with flight software systems but rather hundreds of unfortunately real root causes. Most revolved around failed interface compatibility due to missing or incorrect requirements, changes on one side or other of the interface, poor documentation and communication, and late revelation of the issues.

Jolly also advances the view that software or firmware can no longer be treated as a subsystem, something that is contained to a single element. Because of the absorption of large amounts of complexity by software over the last few decades, the systems engineering teams need software systems engineers, and hardware systems engineers with software backgrounds. He calls for agile yet thorough systems engineering techniques and tools to define and manage the software interfaces because they cannot be handled by the system specification alone or by software subsystem specification attempting horizontal integration with the other subsystems. This situation includes parameter assurance and management. His prediction is that using traditional interface control document techniques will bring a Program to its knees due to the sheer overhead of such techniques (e.g., a 200-page formally adjudicated and signed-off interface control document for every interface). As a remedy, he recommends employing early interface validation via exchanged simulators, emulators, breadboards, and engineering development units. Although the expense of such measures may be daunting, the alternative is that interface incompatibility will not manifest itself until assembly, test, and launch operations, at which time software changes will be the only viable solution. Due to the late stage of development, Jolly expects that neglect of early interface validation will create an inevitable marching-army effect and result in large cost overruns.

Jolly also challenges the notion of procurement of software as black boxes. If the integrator does not understand the software design, its failure modes, its interactions with the physical system and its environment, and how software knits the entire SoS story together, then he expects that software will inevitably be accused of causing overruns and schedule delays. However, the real problem, in his view, is that the integrator did not take account of the real characteristics of the items being integrated.

For software SoS, then, the additional recommendations from Jolly are to ensure that there is an adequate staff of software systems engineers (and hardware systems engineers with software backgrounds), to develop techniques for coordinating many interfaces, to have a greater emphasis on parameter assurance and measurement, to advance away from document-based interface specifications, and to place greater emphasis on interface behaviors, compatibility, and failure modes. As might be expected, these observations bore heavily on the selection of topics addressed in Section B-3.0 of this document.

B-4.2 Modeling Approaches

Not all modeling approaches can handle all the issues involved in software interfaces simultaneously. Some approaches are preferable depending on the relative significance of the issues for the system under consideration, while other modeling approaches are largely incapable of handling certain of the major software interface issues.

In order to address the major software interface issues identified in section B-1.6, the modeling approach must do the following:

- *For Issue 1.6.1:* Where, when, and how software interfaces are addressed
  - Provide a distinction between hardware and software interfaces, or when the distinction would be premature, provide ways to describe interface information that eventually will become relevant to software.
  - Allow associations between functions, physical components, and behaviors.
  - Identify an epoch in the development life cycle where interfaces are addressed.
  - Allow description of functional, physical, and behavioral hierarchies.
- *For Issue 1.6.2:* Software interfaces are driven by the system problem and boundary
  - Identify an epoch in the development life cycle where the system boundary is defined.
  - Provide a method for describing the system problem and boundary.
  - Enable system behavior description and behavioral decomposition.
- *For Issue 1.6.3:* Good software interfaces may not align with good physical interfaces
  - Provide a distinction between hardware and software interfaces.
  - Provide a method for associating functions or software with physical components.
- *For Issue 1.6.4:* Sometimes conceptual spaces for software interfaces need definition
  - Allow nonhierarchical decompositions of systems.
  - Provide methods to describe a wide range of concepts beyond those defined in the method.
  - Allow a broad or even arbitrary range of conceptual spaces in which software interfaces occur.
- *For Issue 1.6.5:* Software interfaces can create an illusion of separation
  - Allow nonhierarchical decompositions of systems.
  - Provide a language to express connections between disparate conceptual spaces.
  - Provide methods to evaluate significance of connections.
- *For Issue 1.6.6:* Sometimes software interfaces show up early at a high level
  - Provide a distinction between hardware and software interfaces
  - Identify an early epoch in the development life cycle where interfaces are addressed
- *For Issue 1.6.7:* Software interfaces are always part of a system function
  - Provide a method for expressing functions.
  - Provide a method for associating functions with physical components or software.
  - Allow description of both functional and physical hierarchies.
  - Provide a method for describing the system problem and boundary.
- *For Issue 1.6.8:* Software interfaces may need computer-aided engineering
  - Provide machine-interpretable representations of systems.
  - Provide a mechanism for machine-based reasoning about system descriptions (if not in the modeling approach, in some of the commercially available tools).
- *For Issue 1.6.9:* Software interface engineering relies on the engineering environment

- Provide methods (not just languages) that recognize the major software interface issues.
- Be associated with a substantial commercial market of tools related to the modeling approach.

- *For Issue 1.6.10:* Software interface complexity won't go away on its own
  - Provide a mechanism for expressing complexity of software interfaces.
  - Provide a mechanism for evaluating and comparing complexity of software interfaces between alternate system specifications.

A comparison of some widely known modeling approaches in their ability to address the key software interface issues appears in Table B-4.3-1.

B-4.3 Engineering and Management Processes

The engineering and management processes should handle:

- Management of numerous stakeholders' satisfaction (or dissatisfaction)
- Management of multiple governing authorities, development organizations, operations organization, and interacting developments
- Decision mechanisms that are efficient when spanning multiple projects
- Accurate estimation of costs for large systems
- Mitigation of cross-cutting risks, not just single-development risks
- Aggregation of information using authoritative and accurate sources of information from component systems
- Efficient communication for SoS

The processes should emphasize clarity of the system problem, especially goals associated with emergent behaviors.

The processes should recognize software interface characteristics as including issues that are different from other kinds of interfaces. The processes should emphasize the component interactions inherent in software interfaces, not merely a provision for receipt of products. Also, the processes should recognize that software is responding to high-level system goals and will be engineered at both high and low levels, not merely used to connect horizontally between system components by means of software interfaces.

The processes should obtain and reason about complexity metrics for software interfaces. Interpretation of metrics is a challenge. One seeks to be able to decide the truth or falsehood of statements like "this interface is more complicated than it needs to be," or "this interface neglects a real complexity that it should handle," or "this complexity is necessary but is more than can be handled." Traceability from the system problem and boundary is necessary to answer such questions, as is an awareness of the organization's ability to handle certain levels of complexity. But these alone are not sufficient; there may be more or less complex ways to solve the system problem. This factor requires looking into alternate solutions, and comparing their complexity on a fair basis. Complexity is not routinely measured at present, so NASA as an organization lacks calibrations of how much complexity it can handle. Measurements, when instituted, will provide an ongoing opportunity for calibration. In the early phases of such a measurement program, only comparisons within a project will be enabled as to the complexity differences between interfaces. Though limited, even this simple step would be useful for resource assessments and would provide incentives to seek alternate solutions to comparatively complex

interfaces. As time goes on, calibrations of organizational capability can be made. Some consideration should be given to qualifying an organization as able to handle particular levels of software interface complexity.

A significant issue for interface complexity measurement is that there is a risk of interfaces being made to look simpler than they really are by means of incompletely or sloppily executed processes. Therefore, metrics are needed to indicate the degree and quality of process execution.

There is also a risk of interfaces being made more complex than they need to be at the whim of the engineer [ref. 47]. Therefore, processes need metrics to indicate the degree of complexity inflation and the degree of traceability to the system problem.

Provisions should be made to examine and make clear choices about the kinds of information that will be included in software interface descriptions. This information should address the conceptual spaces and interface sets involved; refer to the options included in Attachment A.

Configuration management must handle coordination and synchronization on both sides of interface and between higher and lower level models.

The overhead of document centrism should be avoided if possible. Indeed, Jolly [ref. 1] points out that just the document-based overhead can bring projects to their knees. Instead, use model-based control mechanisms, and control the data instead of documents. If there are artifacts created outside the modeling environment, they should be model-based artifacts. The data flow between organizations and modeling environments should be seamless, that is, able to exchange relevant modeling information without manual intervention and with synchronization delays that are small compared with the pace of change.

Methods for specification should rely on models rather than on hand-tailored documents.

Clear decisions are needed about what will and what will not be engineered by the Program or project office.

The processes should include parameter assurance and management for the emergent characteristics at the SoS level to be engineered by the Program or project office.

Table B-4.3-1. Comparison of Some Modeling Approaches as Applied to Software Interfaces

(See Section B-1.6 for summaries of the issues for the corresponding section number. A "y" in the table indicates the approach provides substantial treatment of the issue; an "n" indicates no substantial treatment.)

	Software Interface Issue									
Modeling Approach	B-1.6.1 Where, When, and How Software Interfaces Are Addressed	B-1.6.2 Software Interfaces Are Driven by the System Problem and Boundary	B-1.6.3 Good Software Interfaces May Not Align with Good Physical Interfaces	B-1.6.4 Sometimes Conceptual Spaces for Software Interfaces Need Decades	B-1.6.5 Software Interfaces Can Create an Illusion of Separation	B-1.6.6 Sometimes Software Interfaces Show Up Early at a High Level	B-1.6.7 Software Interfaces Are Always Part of a System Function	B-1.6.8 Software Interfaces May Need Computer-Aided Engineering	B-1.6.9 Software Interface Engineering Relies on the Engineering Environment	B-1.6.10 Software Interface Complexity Won't Go Away on Its Own
Functional Modeling	n	y	n	n	n	y	y	y	n	y
Business Process Model and Notation (BPMN)	y	y	n	n	n	y	y	y	y	y
Integrated Computer Aided Manufacturing Definition for Function Modeling (IDEF0)	n	y	n	n	n	y	y	y	n	y
Integrated Definition for Information Modeling (IDEF1X)	n	n	n	y	n	y	n	y	n	n
Integrated Definition for Simulation Modeling (IDEF2)	n	y	n	n	n	y	n	y	n	y
Integrated Definition for Process Description Capture Method (IDEF3)	n	y	n	y	n	y	y	y	n	y
Integrated Definition for Object-oriented Design (IDEF4)	y	n	y	y	y	y	y	y	n	y
Integrated Definition for Ontology Description Capture Method (IDEF5)	n	n	n	y	y	n	n	y	n	n
Integrated Definition for Design Rationale Capture (IDEF6)	n	y	y	n	n	n	n	y	n	y
Integrated Definition for Human-system Interaction Design (IDEF8)	y	y	n	y	n	y	n	y	n	y
Integrated Definition for Business Constraint Discovery (IDEF9)	n	y	n	n	n	n	n	y	n	y

Modeling Approach	Software Interface Issue									
	B-1.6.1 Where, When, and How Software Interfaces Are Addressed	B-1.6.2 Software Interfaces Are Driven by the System Problem and Boundary	B-1.6.3 Good Software Interfaces May Not Align with Good Physical Interfaces	B-1.6.4 Sometimes Conceptual Spaces for Software Interfaces Need Definition	B-1.6.5 Software Interfaces Can Create an Illusion of Separation	B-1.6.6 Sometimes Software Interfaces Show Up Early at a High Level	B-1.6.7 Software Interfaces Are Always Part of a System Function	B-1.6.8 Software Interfaces May Need Computer-Aided Engineering	B-1.6.9 Software Interface Engineering Relies on the Engineering Environment	B-1.6.10 Software Interface Complexity Won't Go Away on Its Own
Integrated Definition for Network Design Method (IDEF14)	n	n	y	n	n	y	n	y	n	n
Functional Flow Block Diagrams (FFBD)	n	y	n	n	n	y	y	y	n	y
Systems Modeling Language (SysML)	y	y	y	y	y	y	y	y	n	y
Structured Analysis and Design Technique (SADT)	y	y	n	n	n	y	n	y	n	y
Object-oriented Systems Engineering Method (OOSEM)	y	y	y	y	y	y	y	y	y	y
Department of Defense Architectural Framework (DODAF)	n	y	n	n	y	y	y	y	y	y
The Open Group Architecture Framework (TOGAF®)	y	y	y	y	y	y	y	y	y	y
Zachman Framework	y	y	y	y	n	y	y	y	y	y
Hatley-Pirbhai	y	y	n	n	n	y	y	y	n	y
Agent-based Modeling	n	n	n	n	y	n	n	y	n	y
Data Modeling	y	y	n	n	n	y	y	y	n	n
Entity Relationship Modeling	n	n	y	y	y	y	y	y	n	y
Mathematical Modeling	n	n	n	n	n	n	n	y	n	y

The scope of process execution should focus on definition and realization of emergent characteristics at the SoS level, including examination and treatment of emergent characteristics associated with both component and software interface failure modes.

Provisions should be made to examine and make clear choices about suboptimization of component characteristics in order to optimize emergent SoS-level characteristics.

An assessment about the organization's capability to handle complexity should be made. If the capability is not adequate for the project under consideration, the complexity of the project should be reduced as discussed elsewhere in this report. If the complexity cannot be reduced, steps should be taken to improve the organization's capability. Examples of steps that could be taken are acquisition of more capable personnel, training of personnel available to avoid or handle complexity, balancing workforce to the complexity concentration in different parts of the system (possibly including reducing the workforce if its size is contributing to complexity), reducing the number of concurrent works in progress, improving tools to provide special views or scripts that make handling complexity easier, measuring complexity, keeping records of successes and failures, or introducing process management capabilities similar to those used in the upper levels of CMMI.

An assessment should be made about the organization's capability to make sufficiently accurate estimates of resources needed. If the capability is not adequate, reserves should be identified to make up for estimate inaccuracy. If reserves cannot be obtained, steps should be taken to improve the organization's capability. Examples of steps that could be taken are adoption of formal estimating practices, keeping records of estimates and actuals, and devoting sufficient effort to defining the software interfaces before their cost is estimated. When considering the cost of improved estimation, particular attention should be given to the possibility that undetected interface incompatibility will manifest itself late in the life cycle, creating a marching-army effect and huge cost overruns.

An assessment should be made about the completeness, correctness, and clarity of the requirements on SoS-level emergent characteristics associated with software interfaces, both external and internal. If the requirements are not adequate, early iterations of the definition process should be repeated until adequate requirements are available.

Processes should strive for early discovery of software interface issues and should not defer discovery until late design-cycle phases or integration. Early upstrokes of the product realization branch of the system engineering Vee are essential. Interface designs should be validated early using software stubs, virtual integration, simulations, simulators, emulators, breadboards, and engineering development units. This includes interface design validation with the environment. Particular attention should be paid to components that are acquired as black boxes, if they have software interfaces. It is not reasonable to expect components acquired this way to fit into the overall system unless their design, failure modes, and interaction with the physical components and the environment are understood and related to the overall picture of how software contributes to solution of the system problem.

Provision should be made to keep system designs and software designs mutually consistent by means of two-way information exchange between the systems engineering team and discipline or component teams.

Provision should be made to detect and respond to changes on one side or other of the software interfaces. The processes must be agile enough to respond to the wider system-level ramifications of changes as they are made, especially in early phases, rather than pretending that software interactions are purely local, or pretending too soon that the software interface descriptions are fixed. The tendency of software interfaces to carry information that is needed by components other than the two directly involved in a particular interface is a fundamental issue. Change-management processes that can coordinate many interfaces simultaneously are needed.

Program and project system engineering teams should include software system engineers and hardware system engineers with software backgrounds.

A process should exist to distinguish between overruns that are created by (a) poor understanding of the needed emergent characteristics of the system as a whole, (b) poor design of emergent characteristics, and (c) poor realization of component characteristics needed to support emergent characteristics. The process could be as simple as asking four questions in the following order when the overrun is discovered:

- Which emergent system characteristic is associated with this overrun?
- Is this emergent characteristic clearly represented in the system requirements?
- Is this emergent characteristic clearly treated by the design?
- Do the realized components have the characteristics needed to support this emergent characteristic?

The answers to these questions will give guidance as to where effort needs to be applied to resolve the overrun with the least impact; it is not reasonable to expect that the later questions can be corrected without resolving the earlier ones.

Ultimately, there are practical limits to how much can be modeled. Limits should be set based on the need to represent the desired emergent characteristics and based on estimation of risk (significant risks deserve priority in the modeling choices). There are also practical limits on the integration of separate models, so similar criteria should apply.

B-4.4 Integration, Verification, and Validation

A desire for the earliest feasible verification and validation is a significant driver on integration, verification, and validation processes. The desire is based on the observation that it costs less to correct defects earlier in the process of creating a product, giving rise to the widely-recognized 1-10-100 rule. This is a heuristic that states that the cost of correcting a defect increases by a factor of 10 at each step in the production process.

Model-based integration, verification, and validation can be done earlier than traditional methods because the model-based approaches do not have to wait until physical samples, prototypes, or preproduction units are manufactured. Model-based integration of lower-level components can be done in a virtual manner by assembling suites of analysis or executable models describing lower level components to create an analysis or executable model of the end product. Verification can be done between early decompositions of requirements and models of components to confirm the elements are designed as they should be. Validation can be done by deriving aggregated characteristics of the system from the collection of models of the components.

Such practices, when executed on models, can be viewed as early upward passes through the product realization side of the System Engineering Vee. Depending on which intermediate products are chained together, earlier or less complete or later or more complete discovery of defects can be accomplished [ref. 32, Section V]. The NDIA applies the term "virtual integration" to the practice of early integration of models [ref. 30, p. 18]. They point out that virtual integration is a critical strategy both for containing technical risks involving the unknowns associated with the introduction of new technologies in new systems designs and for containing Program risks resulting from the technical risks as well as programmatic unknowns, such as requirements stability, funding, and availability of shared resources. Virtual integration mitigates these risks and provides protection against the most devastating problems arising from early-stage defects: solving the wrong problem, picking the wrong solution, inadequacies in specifications of components, and unforeseen emergent behaviors.

Desires for integration, verification, and validation typically shift as the project matures to focus more on completeness of evaluation, speed, and economy. These desires are readily addressed by model-based techniques if they were started in the early stages of the project. For example, test procedures can be created quickly by simple substitutions of test equipment for the environment in the system behavior models. Automated testing becomes possible if the system models have been created in machine-usable form. It can be simple to measure the extent of verification or validation coverage by tracing from plans and procedures back to the original requirements if the necessary links have been kept. Also, it is less likely that serious troubles will be found in late stages of verification, if early defects have already been corrected by early virtual integrations.

Regarding the cost of verification, Karban (ref. 39) reports that there is an emerging opinion at the European Southern Observatory that consistency and correctness across artifacts is a significant driver on the cost of verification. Highly consistent and correct artifacts make verification simple; greater inconsistency or error leads to a greater verification effort. MBSE can probably assist in ensuring consistency and correctness, though in the ESO opinion this has yet to be proven. For NASA in particular, Carvalho *et al.* [ref. 37] and Bell [ref. 38] report studies that found artifact inconsistency rates in real projects can be surprisingly high, with numerous cases showing discrepancy rates in excess of 40 percent. Both authors report that introduction of model-based techniques to detect and correct discrepancies improved the consistency of artifacts substantially.

To take advantage of the opportunities that model-based integration, verification, and validation offer, some adjustment of the integration style may be needed. That is, a bottoms-up style is often applied in current NASA practice, in which the system engineers identify low-level components from a catalog or early development effort, and then assemble the system in tiers from the bottom up. This style is emphasized in structured systems engineering methods and may be promoted by some interpreters of the NASA Systems Engineering Processes and Requirements [ref. 23] (however, the requirements do recognize the use of phase-appropriate products, including models). The disadvantages of this approach are that of delay, the necessity of expending resources to design and fabricate lower level components, and the possibility of lost investment if defects are eventually found. The remedy may be to practice top-down integration (described below) in early phases, and then shift to bottoms up in the later phases. Some guidance about how to connect processes for this purpose appears in reference 32.

Another style currently in use at NASA is that of pairwise "self-integration," possibly with a third party as the leader. In this style, owners of components negotiate details of interaction, undirected or loosely directed by the SoS owner. This can work well if the details have little consequence for emergent SoS goals, but left to itself may not uncover problems arising from early-stage defects simply because the lower level component owners do not know what they should be looking for. Thus, this style would tend to blunt the benefits of virtual integration, though the mere act of attempting integration of any kind may have a way of bringing mismatches into view. Some centralization of expectations and concern checking may need to be added to recover model-based benefits from this style.

Undirected integration, or "self-assembling" systems, is an integration style that is emphasized by the World Wide Web and the Internet Consortium. In this style, owners of components, or the components themselves, discover each other and interact to create systems they desire, possibly using global conventions of interaction. The question arises as to who is "responsible" for model-based virtual integration in this style. Typically, there is an ecology in which researchers are exploring integration questions in advance of the main developer group, driven by market demands for new features from the self-assembling system. Or, this may be a non-issue if the cost of an implement-fail-correct cycle is trivial.

The top-down integration style is one that is most able to capitalize on the potential of model-based integration, verification, and validation. This style defines top-level stub components early in the design process, assembles a full system using the stubs, and tests system behavior with fledgling behaviors of components. The fledgling behaviors can be tested using increasingly realistic techniques, such as human-powered table-top rehearsals, software stubs using elementary messaging services, or executable system models. This style is emphasized in agile systems engineering methods, where a key goal is to get useful capability into the hands of the customer as quickly as possible in the hope of adapting as quickly as possible to changing requirements or unforeseen consequences. The progressive realism as the stubs are expanded during the development process is a natural fit for the virtual integration concept.

B-4.5 Foundational Capabilities

There are a number of foundational capabilities that are needed for model-based software interface management to work well.

Representation of models. How and where will they be recorded in a machine-usable form? Will they be represented according to a common ontology or a set of compatible ontologies? Will they be represented in a particular language or a set of compatible languages?

Governance of models. Who are the stakeholders, what do they control, how is notification accomplished, and how is agreement accomplished? How are the models themselves controlled?

Model-based acquisition. How can acquisition occur in a model-based rather than a document-based manner and still have the necessary legal attributes? What is the process for model-based acquisition? What are contractors, partners, and collaborators able to use as model-based receivables and deliverables from or to NASA?

Model exchange. How are models are exchanged between owners and users? Is it entire models, abstracted models, artifacts based on the models? What happens when one side or the other of the exchange updates their information later?

Presentation of models. What will be the views, viewpoints, library viewpoints, and correspondence rules for the overall collection of NASA's model assets? Will they be uniform across NASA, or will they be different but compatible?

Use of models in mission development. How can model-based products received or generated be used in design and test environments for risk reduction, design verification, and for early integration and verification and validation (especially assuming partial, incremental deliveries from all of the sources)?

Functional capabilities of tool sets. Within the concept of tool independance, what functional capabilities will NASA expect of its tool set? What should tool vendors be striving to supply for the NASA market?

In the matter of developing these foundational capabilities, there are cross-cutting themes that apply to them all:

- Data integrity
- Meaningfulness and suitability of model application and interpretation
- Data security and access rights
- Steep learning curve
- Culture change
- Providing visibility for model and non-model users alike

These are all in relatively early stages of development for NASA and need further maturation to fully enable model-based software interface management.

B-5.0 Tool Characteristics

Specific recommendations of particular tools for software interface management would be of temporary value at best because the tools and state of the practice are evolving rapidly. Even inside NASA alone, there are many NASA-wide and Center-level initiatives and activities that may be part of the future integration story, in addition to the many initiatives that academic and professional organizations have underway. So, unlike the case for any particular near-term project, it is impossible to predict today the tools and environments that will be useful for various future NASA projects. Nor is it likely that any particular project will stay on one tool set over its lifetime. Successful integration now requires dealing with tool-independent mechanisms and standards for information exchange; thus, approaches need to be flexible as system tools and information exchange standards evolve. It is more likely to be effective to focus on interoperability while allowing the use of different tools and environments.

Tools should be capable of interchanging models between developers to support specification, verification and validation, and integration. Ideally, the interchange would be "seamless," that is, free of burdensome manual operations or undue delay. XMI is a standardized format for model interchange between UML-based tools, but in practice an XMI file exported from one tool is unlikely to import correctly into a different tool. The export or import capabilities should be capable of moving all of the metadata and diagram orientation information as well as core model information. Given that it is common for industrial partners to be involved in software development, it would be helpful to rely as much as possible on industry standards for exchange.

There is a high level of complexity for software interfaces; reviews are impossible without machine-based design checks and dynamic exploration of design. The Constellation Program

integration suffered from the lack of timely and accurate design descriptions, especially in support of key milestone reviews. Tools are needed that address these issues by providing review artifacts that are timely and accurate and that provide views that address all the stakeholder concerns relevant for the project. There may also be a need to adopt different review operations concepts, for instance, for presenters to be tour guides within a modeling environment.

There is a need for tools to calculate complexity metrics for software interfaces, to produce aggregated complexity metrics and measures of organizational ability to handle complexity, and to trace complexity to the overarching system problem.

Mechanisms are needed in the tools to handle multiple concurrent evolutionary states, as routinely arise in real software engineering efforts.

There is a need for standardized information exchange between models used by the Program or project and interfacing models.

There is a need for higher ability to handle complexity in the tools.

One key feature of software interfaces—that they exist in conceptual spaces at multiple levels—raises a requirement on tools: that they must be able to handle many viewpoints simultaneously and be able to keep track of the relationships between entities in different views. At this time, there are no broad agreements on the full range of conceptual spaces that must be handled. There are some common ones (the OSI seven-layer model, SysML diagram types, IDEF, BPMN, various patterns, DoDAF, and many other architecture frameworks), but it is likely that any given project will involve at least some concepts that are unique. For this reason, it is important that tools be able to handle a wide range of viewpoints and have at least some ability to define project-unique viewpoints. There is also a need to be able to "stitch" together composite views.

Documentation consistency is a challenge. It is necessary that tools be capable of producing consistent documentation, at least in snapshot form, across many views and models simultaneously. These all should be based on single sources of authoritative information.

Tools already exist to autogenerate interface software directly from models.

The differences between the physical and software boundaries should be identified. Models allowing the representation of multiple conceptual spaces are necessary.

B-6.0 Summary

Described in this report is a strategy for managing software interfaces successfully on large Programs, comprising a set of objectives to be achieved at various points during the NASA Program or project life cycle. The proposed strategy recognizes key characteristics of software: that software exists in a conceptual realm where interfaces are not necessarily bound by physical constraints; that software interactions can occur in multiple conceptual spaces contemporaneously; that good software interfaces do not always align to physical equipment boundaries; and that the kind of information needed for software interface engineering focuses heavily on aggregated functions to be performed, information exchanges, coordination of behaviors, and management of system states, conditions, and performance.

Additionally, ten major issues regarding software interfaces were identified, and these issues were addressed with recommended activities during the system development life cycle. These

are codified in a software interface management checklist given in Attachment B. Observations are provided about methods and practices and about tool characteristics needed to support the software interface engineering effort. Each of these areas is summarized below.

B-6.1 The Major Issues

Every project decides *where, when, and how software interfaces are addressed*. The NASA project management and system engineering standards say little about when software interfaces are to be addressed, which means that decisions must be made locally to each project. Such decisions may be made by default or too late, sometimes driven by a fallacious belief that software lags hardware.

For some aspects of the software interface challenge, it is actually necessary to *start software interfaces sooner than hardware*. The underlying reason is that software interfaces are driven by the project complexity and by the system boundary issues. Therefore, they must be considered in conceptual stages.

Sometimes *software interfaces may show up at higher levels in the system hierarchy* than the level at which they are eventually implemented. This may happen because it is obvious that an interface with the environment is going to be a software interface because a high-level allocation of software function is made between major physical components, because interfaces between high-level functions are identified early (and eventually deployed on a software system), or because the system itself is a software system. The traditional view that software interfaces should not be discussed in the early stages may not be effective or even correct in any of these cases. Some guidance on when and when not to discuss software during conceptual design should be added to NASA systems engineering policies and handbooks; indeed, NASA policies require awareness of software early in the life cycle.

Good software interfaces may not always align with good physical interfaces. Good interfaces should be simple, facilitate delegation, solve the system problem, match what the interfacing parties can supply or use, etc. Application of these criteria for software may conflict with good hardware interfaces. Typical solutions are software clients, drivers, services, scripts, application program interfaces, etc.

Software interfaces can create an *illusion of separation* because a single software interface can exist in multiple conceptual spaces. Examples of these spaces are the seven layers of the OSI model or the way in which an email message can exercise application, middleware, operating systems, intervening communications layers, and hardware. The overlap between these conceptual spaces can create unanticipated coupling between things that are believed to be separated. Therefore, separation of components in software can be an illusion, more of an emergent property that must be created through design effort than something that arises simply because a boundary has been drawn between components.

Interfaces are always *part of a system function* because every part of a system should be addressing one or more functions of the system. Software interfaces are no exception. For software interface engineering to occur, it is necessary to track the software producers and consumers of interface information, both in the sense of to which functions a software interface is contributing and how each function involved depends on the software interface.

Typically, there are a large number of system functions in which any given software interface is participating, and there is a high potential for complex, multilayered conceptual spaces in which

software interacts. It is common that the complexity exceeds that which can be managed within human capability alone for reasonable cost, so typically *software interfaces need computer-aided engineering*.

The *complexity of software interfaces* tends to be higher than other kinds of interfaces.

Interfaces in hardware tend to be simpler than in software because physical constraints discourage complexity. For physical interfaces, any added complexity greatly increases the difficulty of design and construction—costs are immediate and obvious. Also, physical separations tend to reduce or simplify the degree of interaction between physical components. Software has few physical constraints, and logical connections are easy to introduce.

Interfaces between software components are often "invisible," or not obvious. It is easy to make anything depend on anything else. *Adding complexity to software interfaces is easy in the short run*, and cost consequences are delayed, so there is a relatively flat route to complexity of software interfaces.

Software interface complexity won't go away on its own. The barriers to creating complex software interfaces are low; in fact, it may even be easier to create complex interfaces than simple ones.

There is a *negative tradeoff between software module complexity and software interface complexity*. Separating a complex piece of software into smaller modules will make the modules simpler but will increase the number of software interfaces. Those interfaces will become more complex to handle the larger number of smaller components.

There are *limits to how much complexity can be handled safely*, even given the use of machine aids. Therefore, it is necessary to set limits. Effective strategies for limiting complexity may be:

- Introduction of discipline and training to minimize or mitigate complexity.
- Evaluation of the software interface complexity associated with choices in the architecting process.
- Making hard choices to simplify the system under consideration, removing drivers on complexity by removing system features, finding alternate ways to control hazards other than multiplicity or software controls, and reducing the required degree of flexibility.

B-6.3 Areas of Development Needed

B-6.3.1 Project Management Enhancements

The following areas in project management need to be improved to support model-based software interface management:

- Numerous stakeholders' satisfaction (or dissatisfaction) must be managed.
- Development organizations and interacting developments must be managed—and it isn't a hierarchy.
- Independent operational components need coordination.
- Independent governing authorities need coordination.
- Decision mechanisms must be much more efficient than for single systems.
- Provision must be made to mitigate cross-cutting risks, not just single-development risks.
- Estimation must be more accurate than for single systems.
- Communication must be more efficient than for single systems.

- Software or firmware can no longer be treated as a subsystem—it IS the system (or the SoS).

B-6.3.2 Systems Engineering Enhancements

The following areas in systems engineering need to be improved to support model-based software interface management:

- Requirements for the SoS have to focus on the emergent characteristics, both desired and undesired, for instance, to recognize which engineering iteration applies to particular module features or to allow iterations to exist asynchronously.
- Mechanisms are needed to handle multiple evolutionary states simultaneously.
    - Techniques are needed for coordinating many interfaces, necessitating an advance away from document-based interface specifications and greater emphasis on interface behaviors, compatibility, and failure modes.
- Systems engineering at the SoS level must stay out of components.
    - Much greater emphasis is needed on interactions of components and on SoS parameter assurance and measurement.
        - Interacting systems, disciplines, and software and hardware subsystems must be successfully integrated in order to analyze safety and other SoS properties.
        - Verification of safe system interactions for critical mission activities (e.g., launch aborts) in an SoS environment is particularly challenging.
- Suboptimization of components must be accepted to optimize SoS characteristics.
- The ability to handle complexity must be greater than for a single system.
- Systems engineering teams need software systems engineers and hardware systems engineers with software backgrounds.

Because there are limits to how much complexity can be managed successfully, even with model-based techniques, decisions must be made about how much complexity to engage. In making such decisions, it is imperative to know the capability of the affected organizations to handle the complexity contemplated. NASA should start tracking measures of software interface complexity for projects attempted and maintain a record of successes and problems encountered. At this time, there is not a single well-recognized set of software interface complexity measures, although a number of possibilities exist. Improvements in definition and measurement of complexity are needed to enable the necessary tracking of organizational capability.

B-6.3.3 Tool Enhancements

A variety of tool improvements are needed. To be able to handle the multiple conceptual spaces inhabited by software interfaces, tools must be able to handle many viewpoints simultaneously and be able to keep track of the relationships between entities in different views. Tools for software interface engineering should handle a wide range of viewpoints, including local specializations, and have the ability to stitch together composite views from different spaces. Handling of multiple concurrent evolutionary states is necessary. Calculation of complexity metrics from models of software interfaces is necessary, as is tracing of complexity to the overall system problems to be solved. A capability for seamless exchange of models between domains would be useful in reducing labor, errors, and delays in communicating software design information that is related to information defined in other disciplines.

There is not yet a specialized ontology for software fundamentals equivalent to what is available for business, systems engineering, and general fundamental phenomena. Until that is available, each project will be developing its own solutions for expressing the conceptual spaces in which software interfaces are defined. As this is a pivotal issue for software interfaces and there will be much duplication of work between projects to address it, an Agency-level solution should be pursued.

It would be beneficial to the software interface effort to have an efficient capability for general information sharing between models in different disciplines. For model information sharing to be efficient, the models themselves must be interoperable based on standards. When models are developed in different environments, interoperability will require mechanisms to transform models from one environment to another. The model interoperability standards do not presently exist, nor do the model transformation mechanisms, making this a needed area of development.

There is also a need to address configuration and change control, security, and access control for models, tools, and simulation data. Concerning this issue and the information exchange issues mentioned above, NASA should pursue a collaboration with INCOSE. This is likely to be fruitful because the INCOSE Systems Engineering Vision for 2020 includes domain-specific modeling languages and visualization; modeling standards based on a firm mathematical foundation; extensive reuse of model libraries, taxonomies, and design patterns; standards that support integration and management across a distributed model repository; and highly reliable and secure data exchange via published interfaces. NASA should be able to capitalize on and contribute to the shared development of these topics.

Model transformation to support software interface development would benefit from development of solutions to integrate system modeling with general computational services (e.g., MATLAB®, Mathematica®, Maple®), high-end simulation or analysis tools (e.g., Modelica®, Phoenix® ModelCenter, ARENA®), and, in particular, specialized correctness checkers or provers (e.g., SPIN, Java™ Pathfinder).

B-6.4 How can Model-centrism Help?

In the project management arena, there are immediate opportunities to improve several areas using model-centrism. It is possible to improve efficiency of communications and decision mechanisms by incorporating work flows into a model-based environment, with increased reliance on data in the models and decreased reliance on documents, boards, and panels. Model-centrism can make it possible to recognize integrated software SoSs as a primary system development, so that it becomes possible to map from software deployment to physical systems in the models, not the other way around (a significant advantage for systems dominated by software complexity, though perhaps not for systems with minimal software). Model-centrism can address Jolly's observation that software cannot be treated as a subsystem. And, it is currently possible to capture Level 1 verifications in the early model (they should, of course, be known by that time) and use them to drive verification and validation plans downstream. Application of model-based techniques can probably improve the consistency of project artifacts, thereby lowering the cost of verification. Finally, model-based virtual integration can provide protection against the most devastating problems arising from early-stage defects: solving the wrong problem, picking the wrong solution, inadequacies in specifications of components, and unforeseen emergent behaviors.

Development opportunities on the near horizon for project management include improved management of stakeholder (dis)satisfaction by using specialized viewpoints to address their concerns from the beginning. With small developments, it should be possible to improve management of development organizations, interacting developments, operational components, and governing authorities by extending models to include organizations, their interfaces, and functions; indeed, this is already included in the NASA Data Architecture Framework used by the Human Exploration and Operations Mission Directorate [ref. 31]. It should be possible to improve estimation accuracy by relying on models both up front and downstream because there are model-based estimation technologies already available for software.

In the systems engineering arena, immediate opportunities are available to use models to express realization of SoS-level emergent characteristics, for instance, by relating functional models of the emergent characteristics to physical models. Models and supporting tools can handle more complexity than can humans, even if the models do not have infinite capacity to handle complexity. Model variants can be used to handle multiple evolutionary states simultaneously. It is practical to capture interactions of components, to capture parameter allocations and estimates, to capture complex scenarios in models, and to interrogate the models for verification. Top-down integration techniques based on evolving fidelity of models are practical, to facilitate early discovery of interactions. It is also practical to increase reliance on models as specifications for interfaces, to decrease reliance on documents, and thereby reduce the risk associated with burdensome interface management practices.

Development opportunities on the near horizon for systems engineering include the use of models to trace relationships between emergent characteristics and component requirements, the capability to develop viewpoints to justify suboptimization of components to achieve required SoS characteristics, and the integration of models with varying degrees of maturity.

B-6.2 Handling the Major Issues during the System Development Life Cycle

Proposed methods to handle the major issues in a strategic manner at each stage of the system life cycle were presented. The strategy begins with an up-front examination during the concept study phase of the critical questions that must be answered to create an adequate foundation for software interfaces. Definition of the system problem to be solved at the highest level is critical, with special attention to obtaining the kind of information needed downstream for software interface work. Also, it may be necessary to define the conceptual spaces within which external software interfaces will exist.

The strategy then proceeds to realize desired system-level characteristics through concept development and preliminary design, with careful consideration of system physical and functional partitioning, in some cases identifying software at the highest level. Potential pathologies of software interfaces should be evaluated and, if necessary, corrected using the techniques identified in the text. The design strategy continues with elaboration of interface behaviors to a degree that greatly reduces the chances of late discovery of software incompatibilities downstream. Consideration should be given to situations where the software interfaces are different from the physical boundaries.

Later stages ensure discipline to and verification of software interfaces using the constructs developed in the early stages. System training, operation, and reuse are enhanced downstream using the software interface products developed along the way.

B-6.5 Closing Comment

Successful software interface management at the SoS level will depend on making wise choices regarding how much complexity to engage, starting management of software interfaces during the conceptual phases of the Program or project life cycle, using effective methods or practices or tools, and addressing the major issues of software interface management at each stage of the system development life cycle. Model-based approaches offer excellent solutions to managing interface complexity at all levels of development. There are aspects of the recommended strategy that should be enacted during the earliest stages of a Program in order to be effective; even though specific software interfaces may not be identified until later design stages, the seeds for their success must be sown upstream.

B-7.0 Acronyms

21 CGSP	21st Century Ground Systems Program
AADL	Architecture Analysis and Design Language
AFT	Architecture Framework Tool
ARC	Ames Research Center
ATL	Atlas Transformation Language
BPMN	Business Process Model and Notation
CAD	Computer-assisted Design
CAE	Computer-aided Engineering
CASE	Computer-aided Software Engineering
CDRL	Contract Data Requirements List
CMMI	Capability Maturity Model Integration
CoFR	Certification of Flight Readiness
COTS	Commercial-off-the-Shelf
CSCI	Computer Software Configuration Item
CVA	Common Vehicle Architecture
DEM	Data Exchange Message
DoD	Department of Defense
DoDAF	Department of Defense Architecture Framework
EA	Enterprise Architect
ECSAM	Embedded Computer System Analysis and Modeling
ESO	European Southern Observatory
EVA	Extravehicular Activity
FEA	Finite Element Analysis
FDIR	Fault Detection, Isolation, and Recovery
FFBD	Functional Flow Block Diagram
FS&GS	Flight Systems and Ground Support
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
ICD	Interface Control Document
IDEF	Integrated Definition
IDL	Interface Description Language
IFA	Integrated Functional Analysis
IMA	Integrated Mission Analysis

INCOSE	International Council on Systems Engineering
iPAS	Integrated Power, Avionics, and Software
IRD	Interface Requirements Document
JPL	Jet Propulsion Laboratory
KDP	Key Decision Point
LGPL	Lesser General Public License
MBASE	Model-based Architecture and Software Engineering
MBE	Model-based Engineering
MBSE	Model-based Systems Engineering
MCR	Mission Concept Review
MDA	Model-driven Architecture
MDR	Mission Definition Review
MODAF	Ministry of Defence Architecture Framework (British)
MPCV	Multi-Purpose Crew Vehicle
NAS	National Airspace System
NDIA	National Defense Industry Association
OCE	Office of the Chief Engineer
OMG	Object Management Group
OOSEM	Object-oriented Systems Engineering Method
OPM	Object-process Methodology
OSI	Open Systems Interconnection
PDF	Portable Document Format
QVT	Query/View/Transformation
RUP SE	Rational® Unified Process for Systems Engineering
SADT	Structured Analysis and Design Technique
SAM	System Architecture Model
SBU	Sensitive but Unclassified
SCXML	State Chart Extensible Markup Language
SDR	System Design Review
SE&I	Systems Engineering and Integration
SLS	Space Launch System
SMAP	Soil Moisture Active and Passive
SoS	System of Systems
SRR	System Requirements Review
STORM	System for Tracking Operational Readiness for Missions
SysML	Systems Modeling Language
TOGAF®	The Open Group Architecture Framework
UAS	Unmanned Aircraft System
UML	Unified Markup Language
UPDM	United Profile for DoDAF and MODAF
XMI	XML Model Interchange
XML	Extensible Markup Language

B-8.0 References

1. Jolly, S., "Is Software Broken," *NASA Ask Magazine*, No. 34, Spring 2009, pp. 22–25.
2. Hammer, W., *Product Safety Management and Engineering*, Prentice-Hall, Englewood Cliffs, NJ, 1980.
3. Friedenthal, S., Moore, A., and Steiner, R. *A Practical Guide to SysML*, Elsevier, Burlington, MA, 2009.
4. Leveson, N., *Safeware: System Safety and Computers*, Addison-Wesley, Boston, MA, 1995.
5. Buede., D. M., *The Engineering Design of Systems: Models and Methods*, 2nd Edition, Wiley, Hoboken, NJ, 2009.
6. "NASA Systems Engineering Processes and Requirements," NPR 7123.1A, March 26, 2007. URL: http://nodis.hq.nasa.gov.
7. "NASA Systems Engineering Handbook," SP-2007-6105, Rev. 1, December 2007. URL: http://ntrs.nasa.gov.
8. Yakimovich, D., Travassos, G., and Basili, V., "A Classification of Software Components Incompatibilities for COTS Integration," Presented at the 24th Software Engineering Workshop, NASA Goddard Space Flight Center, Greenbelt, MD, December 1999. URL: http://chess.cs.umd.edu/~basili/publications/technical/T127.pdf, accessed June 26, 2012.
9. Gacek, C., "Detecting Architectural Mismatches during Systems Composition," Doctoral Dissertation, University of Southern California, Dept. of Computer Science, December 1998. URL: http://csse.usc.edu/csse/TECHRPTS/PhD_Dissertations/files/Gacek_Dissertation.pdf, accessed June 26, 2012.
10. Teixeira, N. S. and Pereira e Silva, R., "Component-oriented Software Development with UML," *Proceedings of the Sixth International Conference on Software Engineering Advances,* International Academy, Research, and Industry Association, Curran Associates, October 23, 2011.
11. Breidenthal, J., *et al.*, "Constellation Program Software and Avionics Lessons Learned for SE&I," Internal Memorandum, Jet Propulsion Laboratory, February 4, 2011.
12. Ingham, M., *et al.*, "MBSE in Development: SMAP Pilot Project," NASA Model-Centric Engineering Symposium & Workshop at JPL, Jet Propulsion Laboratory, January 24, 2012.
13. Jenkins, S. and Rouquette, N., "OWL Ontologies and SysML Profiles: Knowledge Representation and Modeling," NASA-ESA PDE Workshop, Jet Propulsion Laboratory, May 19, 2010. URL: http://www.congrex.nl/10m05post/presentations/pde2010-Jenkins.pdf.
14. Bayer, T., *et al.*, "Jupiter Europa Mission Concept Studies: Early Formulation MBSE and Lessons Learned," NASA Model-Centric Engineering Symposium & Workshop at JPL, Jet Propulsion Laboratory, January 12, 2012.
15. Oster, C., "Industry State of Practice: Lockheed Martin: Integrating Models to form a Digital Tapestry," NASA Model-Centric Engineering Symposium & Workshop at JPL, Jet Propulsion Laboratory, January 24, 2012.

16. Jenkins, S., "Model Transformation and Integrated Analysis Report Out," NASA Model-Centric Engineering Symposium & Workshop at JPL, Jet Propulsion Laboratory, Jet Propulsion Laboratory, January 24, 2012.
17. Friedenthal, S., "MBSE State of Practice and Future Directions," NASA Model-Centric Engineering Symposium & Workshop at JPL, Jet Propulsion Laboratory, January 26, 2012.
18. Fluhr, J. H., "IT Infrastructure: Key to Successful Application of Model-Based Systems Engineering on NASA Programs," Presented at the NASA Information Technology Summit, March 5, 2010. URL: http://www.nasa.gov/ppt/482614main_2010_Tuesday_4_Fluhr.Jody.ppt.
19. Somervill, K. and Johnston, S., "Langley Research Center MBSE State of Practice," NASA Model-Centric Engineering Symposium & Workshop at JPL, Jet Propulsion Laboratory, January 24, 2012.
20. Vera, A., *et al.*, "Ames Research Center Support for Model-Centric Engineering," NASA Model-Centric Engineering Symposium & Workshop at JPL, Jet Propulsion Laboratory, January 24, 2012.
21. NASA Office of the Chief Engineer, "NASA Space Flight Program and Project Management Requirements," NPR 7120.5E, August 14, 2012. URL: http://nodis3.gsfc.nasa.gov/displayDir.cfm?t=NPR&c=7120&s=5E.
22. NASA Office of the Chief Engineer, "NASA Systems Engineering Handbook," SP-2007-6105, Rev. 1, December 2007. URL: https://standards.nasa.gov/documents/detail/3315825 or http://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/20080008301_2008008500.pdf.
23. NASA Office of the Chief Engineer, "NASA Systems Engineering Processes and Requirements," NPR 7123.1A, 26 March 2007. URL: http://nodis3.gsfc.nasa.gov/displayDir.cfm?t=NPR&c=7123&s=1A.
24. NASA Office of the Chief Engineer, "NASA Software Engineering Requirements," NPR 7150.2A, November 19, 2009. URL: http://nodis3.gsfc.nasa.gov/displayDir.cfm?t=NPR&c=7150&s=2A.
25. NASA Office of the Chief Engineer, "NASA Software Assurance Standard," NASA-STD-8739.8, July 28, 2004.
26. Maier, M. W. and Rechtin, E., *The Art of Systems Architecting*, 2d Edition, CRC Press, Boca Raton, FL, 2000.
27. Maier, M. W., "Architecting Principles for System of Systems," *Systems Engineering*, Vol. 1, No. 4, 1998, pp. 267–284.
28. Charette, R. N., "Why Software Fails," *IEEE Spectrum*, September 2005. URL: http://spectrum.ieee.org/computing/software/why-software-fails.
29. Lane, J. A. and Boehm, B., "System of Systems Lead System Integrators: Where Do They Spend Their Time and What Makes Them More or Less Efficient?" *Systems Engineering*, DOI 10.1002/sys, Wiley, 7 September 2007.

30. NDIA Systems Engineering Division, "Final Report of the Model Based Engineering (MBE) Subcommittee," National Defense Industry Association, February 10, 2011. URL: http://www.ndia.org/Divisions/Divisions/SystemsEngineering/Documents/Committees/M_S%20Committee/Reports/MBE_Final_Report_Document_%282011-04-22%29_Marked_Final_Draft.pdf.
31. Crain, R., "NASA Program Formulation Framework," NASA Integrated Model Centric Architecture Program, October 11, 2012. URL: https://docs-nen.nasa.gov/cloudrepo/file/3fa34604-f9f9-4649-83d4-a041de688ab3/Program Formulation FrameworkR5.docx.
32. Breidenthal, J. and M. Overman., "Layered System Engineering Engines," AIAA-2009-1885, 6 April 2009. URL: http://arc.aiaa.org/doi/pdf/10.2514/6.2009-1885.
33. Stoewer, J., "Model-based Engineering (MBE) and Model-based Systems Engineering (MBSE) in Context: The Golden Age of Simulation?" NASA Model-Centric Engineering Symposium & Workshop at JPL, Jet Propulsion Laboratory, January 24, 2012.
34. Kahn, O. and Standley, S., reported in Ingham, M., "MBSE in Development: SMAP Pilot Project," NASA Model-Centric Engineering Symposium & Workshop at JPL, Jet Propulsion Laboratory, January 24, 2012.
35. Breidenthal, J., "Constellation Program Software and Avionics Lessons Learned: 418 Systems Engineering Management Summary," NASA Internal Presentation, 8 December 2010, p. 17.
36. DeLaurentis, D., "Model-Centric Analysis for Interdependent Systems," NASA Model-Centric Engineering Symposium & Workshop at JPL, Jet Propulsion Laboratory, January 24, 2012, pp. 11, 16, 17.
37. Carvalho, R. E., "Integrating Engineering Data Systems for NASA Spaceflight Projects," *Proceedings of the IEEE Aerospace Conference*, March 3, 2012. URL: http://ieeexplore.ieee.org/stamp/ stamp.jsp?arnumber=06187344.
38. Bell, D. G., "Data Integrity in NASA Programs and Projects," NASA Integrated Model Centric Architecture Program, July 31, 2012. URL: https://docs-nen.nasa.gov/cloudrepo/file/d800b26b-71e2-455e-bdca-788a353d2ac9/Data Integrity White Paper - NIMA-RPT-002.pdf.
39. Karban, R., "ESO State of Practice: MBSE for Large Telescopes," NASA Model-Centric Engineering Symposium & Workshop at JPL, Jet Propulsion Laboratory, January 24, 2012, pp. 50–51.
40. Crisp, H. E., II, "INCOSE Systems Engineering Vision 2020," INCOSE-TP-2004-004-02, September, 2007. URL: http://www.incose.org/ProductsPubs/pdf/SEVision2020_20071003_v2_03.pdf.
41. Joint Technical Committee ISO/IEC JTC 1, "Information Technology—Open Systems Interconnection Basic Reference Model: the Basic Model," Second Edition, ISO/IEC 7498-1(E), International Organization for Standardization and International Electrotechnical Commission, Switzerland, 1994-11-15; corrected and reprinted 1996-06-15. URL: http://standards.iso.org/ittf/PubliclyAvailableStandards/s020269_ISO_IEC_7498-1_1994(E).zip.

42. De Marco, T., *Structured Analysis and System Specification*, Prentice-Hall, New Jersey, 1979.

43. DoD Chief Information Officer, "DoD Architecture Framework, Version 2.0," Vol. 1, U.S. Department of Defense, August 2010. URL: http://dodcio.defense.gov/dodaf20.aspx.

44. Moody, T., *et al.*, "JSC State of Practice," Proceedings of the NASA MBE Symposium & Workshop at JPL, Jet Propulsion Laboratory, January 24, 2012.

45. The Object Management Group, "CORBA Basics," September 11, 2012. URL: http://www.omg.org/gettingstarted/corbafaq.htm.

46. "Data Item Description: Interface Design Description (IDD)," DI-IPSC-81436A, U.S. Department of Defense, December 15, 1999. URL: http://www.everyspec.com/DATA-ITEM-DESC-DIDs/DI-IPSC/DI-IPSC-81436A_3748/.

47. Dvorak, D., *et al.*, "NASA Study on Flight Software Complexity," NASA Office of Chief Engineer, 2009. URL: www.nasa.gov/pdf/418878main_FSWC_Final_Report.pdf.

48. Bromley, L., "NASA Integrated Model-Centric Architecture (NIMA)," NASA Model-Centric Engineering Symposium & Workshop at JPL, Jet Propulsion Laboratory, January 24, 2012.

49. Bindschadler, D., "MBSE for A Product Line: MOS 2.0," NASA Model-Centric Engineering Symposium & Workshop at JPL, Jet Propulsion Laboratory, January 24, 2012.

50. Dvorak, D., "Intro to JPL State of Practice," NASA Model-Centric Engineering Symposium & Workshop at JPL, Jet Propulsion Laboratory, January 24, 2012.

51. Fuchs, J., *et al.*, "ESA State of the Practice," NASA Model-Centric Engineering Symposium & Workshop at JPL, Jet Propulsion Laboratory, January 24, 2012.

52. Arteaga, R., "Dryden FRC State of the Practice," NASA Model-Centric Engineering Symposium & Workshop at JPL, Jet Propulsion Laboratory, January 24, 2012.

53. INCOSE SE2 Challenge Team for Telescope Modeling, "MBSE for MagicDraw," ESO, HOOD Group, TU Munich, oose GmbH, and GfSE, 2011. URL: http://sourceforge.net/projects/mbse4md/files/.

54. Aguilar, M., "Mitigation of Risks Through the Use of Computer-Aided Software Engineering (CASE) Tools," private communication, NASA Engineering and Safety Center (NESC), January 24, 2012.

55. Doyle, P., "MSFC State of the Practice," NASA Model-Centric Engineering Symposium & Workshop at JPL, Jet Propulsion Laboratory, January 24, 2012.

56. Bucaille, S., "Florida Institute of Technology Spacecraft Systems: A Comprehensive MBSE Program," NASA Model-Centric Engineering Symposium & Workshop at JPL, Jet Propulsion Laboratory, January 24, 2012.

57. Watson, J., "Outbrief for Model Management Workshop," NASA Model-Centric Engineering Symposium & Workshop at JPL, Jet Propulsion Laboratory, January 24, 2012.

58. Ryschkewitsch, M., *et al.*, "Standard for Models and Simulations," NASA-STD-7009, July 11, 2008. URL: https://standards.nasa.gov/documents/viewdoc/3315599/3315599.

59. Conroy, M., "KSC State of the Practice," NASA Model-Centric Engineering Symposium & Workshop at JPL, Jet Propulsion Laboratory, January 24, 2012.

60. Breidenthal, J., "Constellation Program Software and Avionics Lessons Learned for SE&I," NASA Internal White Paper, Constellation Program Office, Johnson Space Center, February 4, 2011.

61. Estefan, J., "Survey of Model-Based Systems Engineering (MBSE) Methodologies," INCOSE-TD-2007-003-01 Rev. B, INCOSE Model Based Systems Engineering Initiative, Seattle, WA, June 10, 2008. URL: http://www.incose.org/productspubs/pdf/techdata/mttc/mbse_methodology_survey_2008-0610_revb-jae2.pdf.

62. NASA Office of the Chief Engineer, "Standard for Models and Simulations," NASA-STD-7009, July 11, 2008. URL: http://standards.nasa.gov/documents/viewdoc/3315599/3315599.

63. Bromley, L., *et al.*, "NASA Integrated Model-Centric Activity (NIMA) Benchmarking Team Final Report," NIMA-RPT-001, August 20, 2012. URL: https://docs-nen.nasa.gov/cloudrepo/file/f8fdd312-2e7c-432b-bcfc-53380dfcaea1/NIMA-RPT-001 Benchmarking Final Report Draft_RevK.docx.

64. Morillo, R., "Constellation Program Software Management Lessons Learned Summary," NASA Internal Presentation, Lesson 376, Constellation Program Office, Johnson Space Center, January 13, 2011.

65. Madden, J., "100 Lessons Learned for Project Managers (Lesson No. 72)," *NASA Ask Magazine*, Issue 14, October 2003. URL: http://askmagazine.nasa.gov/issues/14/practices/ask14_lessons_madden.html.

Attachment A. Options for Software Interface Content

TABLE A-1. OPTIONS FOR INTERFACE CONTENT

OPTION 1 (software to hardware):

1 INTRODUCTION
1.1 IDENTIFICATION
1.2 DOCUMENT SCOPE
1.3 DEFINITIONS
2 8051 OR FPGA INTERFACE
2.1 INSTRUMENT
2.1.1 FULL MEMORY MAP
2.1.2 MEMORY MAPPED I/O
2.2 DREB
2.2.1 MEMORY MAP
2.2.2 MEMORY MAPPED I/O
3 INTERRUPTS
3.1 INSTRUMENT
3.2 DREB
4 RESETS
4.1 COMMANDED WARM BOOT
4.2 COMMANDED COLD BOOT
4.3 WATCHDOG RESET
5 BOOTUP SYNCHRONIZATION
5.1 INSTRUMENT
6 1553 ADDRESSING

OPTION 2 (software to software, layered communication):

1.0 INTRODUCTION
1.1 PURPOSE
1.2 SCOPE
1.3 CHANGE AUTHORITY/RESPONSIBILITY
2.0 DOCUMENTS
2.1 ORDER OF PRECEDENCE
2.2 APPLICABLE DOCUMENTS
2.3 REFERENCE DOCUMENTS
3.0 INTERFACE DEFINITION
3.1 NETWORKS
3.1.1 PHYSICAL AND DATA LINK LAYER IMPLEMENTATION
3.1.1.1 ETHERNET FRAME AND HEADER DEFINITION
3.1.1.2 ETHERNET ADDRESSING
3.1.1.3 ETHERNET TRAFFIC
3.1.2 NETWORK AND TRANSPORT LAYER IMPLEMENTATION
3.1.2.1 INTERNET PROTOCOL VERSION 4 (IPV4) DEFINITION
3.1.2.2 INTERNET PROTOCOL VERSION 6 (IPV6) DEFINITION
3.1.2.3 USER DATAGRAM PROTOCOL (UDP)
3.1.3 DATA EXCHANGE LAYER IMPLEMENTATION
3.1.3.1 DATA EXCHANGE LAYER DESCRIPTION
3.1.3.2 COMMAND DEM
3.1.3.3 TELEMETRY DEM
3.1.4 APPLICATION LAYER IMPLEMENTATION
3.1.4.1 REAL-TIME TRANSPORT PROTOCOL
3.1.4.2 FLIGHT CRITICAL LINK
3.1.4.3 MRD LINK
3.2 SECURITY IMPLEMENTATION

3.3 UNIQUE CHARACTERISTICS OF IRD IMPLEMENTATION
APPENDIX A, ACRONYMS, ABBREVIATIONS AND GLOSSARY OF TERMS
A1.0 ACRONYMS AND ABBREVIATIONS
A2.0 GLOSSARY OF TERMS
APPENDIX B, OPEN WORK
B1.0 TO BE DETERMINED
B2.0 TO BE RESOLVED
APPENDIX C, REQUIREMENT TRACEABILITY MATRIX
APPENDIX D, ORION COMMAND DEFINITIONS
APPENDIX E, TELEMETRY PACKING MAP SETS

OPTION 3 (software to software, services):
3.3 FLIGHT/GROUND INTERFACE
3.3.1 ACTOOLS – AUTOCODE TOOLS
3.3.2 BFS – BUFFERED FILE SERVICE
3.3.3 CMD – COMMAND DISPATCH
3.3.4 CP – COMMAND PROXY
3.3.5 DDI – DEFINITION DATA ITEMS
3.3.6 DMS – DATA MANAGEMENT SERVICE
3.3.7 DTS – DATA TRANSPORT SERVICE
3.3.8 DWN – DOWNLINK
3.3.9 EHA – CHANNELIZED TELEMETRY SERVICE (ENGINEERING, HOUSEKEEPING, AND ACCOUNTABILITY)
3.3.10 EVR – EVENT REPORTING SERVICE
3.3.11 NPM – NON-VOLATILE PARAMETER MANAGER
3.3.12 NVDS – NON-VOLATILE DATA STORAGE
3.3.13 NVFS – NON-VOLATILE FILE SYSTEM
3.3.14 NVMCFG – NON-VOLATILE MEMORY CONFIGURATION
3.3.15 RAMFS – VOLATILE FILE SYSTEM
3.3.16 SEQ – SEQUENCE ENGINES
3.3.17 SPS – STREAM PRODUCT SERVICE
3.3.18 UPL – UPLINK
3.3.19 VID – VIRTUAL ID SERVICE MODULE

OPTION 4 (software combined with other domains and verification):
1. SCOPE
1.1 IDENTIFICATION
1.2 SYSTEM OVERVIEW
1.3 DOCUMENT OVERVIEW
2. DOCUMENTS
2.1 GENERAL
2.2 APPLICABLE DOCUMENTS
2.3 REFERENCE DOCUMENTS
2.4 ORDER OF PRECEDENCE
3. INTERFACE SPECIFICATION
3.1 SPACECRAFT MISSION CHARACTERISTICS
3.1.1 MISSION DESCRIPTION
3.1.2 INTERFACING ELEMENT DESCRIPTION
3.1.3 INPUT DATA FOR MISSION ANALYSES
3.1.4 ORBIT PARAMETERS (WITH TOLERANCES)
3.1.5 LAUNCH WINDOW
3.1.6 POINTING AND SEPARATION
3.1.7 INTERFACE DIAGRAMS
3.1.8 INTERFACE NAME

Attachment B – Checklist of Software Interface Management Activity by Life Cycle Phase

B.1 General – For All Interfaces

- ☐ Identify things that will interact
- ☐ Identify conceptual spaces of interaction
- ☐ Identify interface set
- ☐ Identify material interfaces
- ☐ Identify nonmaterial interfaces
- ☐ Identify boundary regions
- ☐ Identify interior regions

B.2 General - For Software Interfaces

- ☐ Identify type of software entities involved in interface (computer program, computer instructions, physical states of a computer, data manifested in a physical form, sensors, actuators, data, algorithms, processes, languages, rules, decision weights, descriptive supporting information, prescriptive supporting information)
- ☐ Identify differences in location between material and nonmaterial interfaces
- ☐ Identify material manifestation type of interface (physical boundary of a piece of equipment, no material interface anywhere, appear at many material interfaces simultaneously)
- ☐ Identify underlying layers of interaction
- ☐ Check for illusions of separation
- ☐ Describe special nonmaterial concepts associated with interfaces
- ☐ Assess complexity of interfaces
- ☐ Estimate life cycle costs associated with complexity
- ☐ Identify management processes for software interfaces
- ☐ Identify engineering processes for software interfaces
- ☐ Assess ability of processes to handle nonmaterial interfaces

B.3 General - Model-Based Engineering

- ☐ Choose fidelity of models
- ☐ Choose types of models (physical, quantitative, qualitative, or mental)
- ☐ Choose applications of models (definitive, descriptive, or normative)
- ☐ Choose methods of exchanging models (oral, writing, machine)
- ☐ Choose machine aids to test models for completeness, consistency, and accuracy
- ☐ Choose machine aids to exchange information between models and modelers
- ☐ Choose machine aids to maintain consistency between models
- ☐ Choose machine aids to derive system characteristics
- ☐ Assess adequacy of model-based engineering situation (speed, accuracy, range of knowable and controllable characteristics, level of complexity that can be achieved, ambiguity, error, and confusion)
- ☐ Assess time available and resources available

B.4 General - The NASA Program or Project Life Cycle

- ☐ Choose times at which software will and will not be engineered
- ☐ Check for myth of having to wait until hardware is available to design software.
- ☐ Determine proportion of software in the system
- ☐ Determine portion of system problem to be solved by software
- ☐ Determine degree of interaction of software
- ☐ Assess complexity of system problem
- ☐ Define system-level problems to be solved
- ☐ Define system boundary
- ☐ Identify desired and undesired system-level "emergent" characteristics
- ☐ Identify desired and undesired system-level "emergent" characteristics associated with software
- ☐ Assess proportion of system complexity associated with software
- ☐ Check for match between proportion of engineering effort to be devoted to software, and proportion of system complexity associated with software
- ☐ Create partitions in the architecture
- ☐ Identify which compartments in the architecture will involve software
- ☐ Identify which compartments in the architecture will have software interfaces
- ☐ Identify functions that solve the system problem
- ☐ Identify which functions will be allocated to software
- ☐ Identify which functions will be allocated to software interfaces
- ☐ Identify which performance will be allocated to software
- ☐ Identify which performance will be allocated to software interfaces
- ☐ Assess effectiveness of the software system design at solving the system problem
- ☐ Assess effectiveness of the software system design at solving the allocated sub-problems within the system that contribute to solving the high-level problems.
- ☐ Define engineering environment to support software engineering
- ☐ Identify classes of people making up the engineering environment (managers, engineers, acquirers, testers, operators, maintainers, lawyers, financers, trainers, etc.)
- ☐ Identify hardware used for engineering
- ☐ Identify software used for engineering
- ☐ Identify policies for engineering
- ☐ Identify methods of engineering
- ☐ Identify information involved in engineering
- ☐ Assess suitability of the engineering environment to support software engineering
- ☐ Assess suitability of the engineering environment to support software interface engineering

B.5 Issues Watch List

- ☐ Decide where, when, and how software interfaces are addressed
- ☐ Define the system problem and boundary
- ☐ Relate parts of the system problem to the software interfaces as they are created

- ☐ Relate behavior, information, or media exchanges at the system boundary with software interfaces
- ☐ Obtain information to enable engineering of software interfaces
- ☐ Check for hardware information that may eventually lead to software interface design constraints
- ☐ Check that software interfaces are simple, facilitate delegation of the problem to appropriate kinds and sizes of organizations, satisfy all the relevant requirements needed to solve the system problem, and match what the interfacing parties can supply or use
- ☐ Check that software isolate internal details, are robust against failures in either the environment or the system, and facilitate system upgrades, changes, or interoperability
- ☐ Check for conflicts between making good software interfaces and making good hardware interfaces
- ☐ Consider placing software interfaces at a different location than hardware interfaces
- ☐ Assess degree of knowledge possessed by the workforce for the conceptual spaces in which the software interfaces occur
- ☐ Consider describing conceptual spaces for software interfaces explicitly
- ☐ Check for illusions of separation
- ☐ Check for software interfaces that show up early at a high level
- ☐ Identify to which system functions each software interface contributes
- ☐ Identify on which software interfaces each system function depends
- ☐ Assess need to use computer aids for software interface engineering
- ☐ Assess suitability of the personnel, organizations, facilities, methods, processes, policies, information, and tools composing the engineering environment
- ☐ Assess ability of the engineering environment to handle nonmateriality, conceptual spaces, potential for deviation from physical interfaces both in location and inherent properties, the potential illusion of separation, and difficulties of communication
- ☐ Assess negative tradeoff between software module complexity and software interface complexity
- ☐ Assess how much complexity can be handled safely, even given the use of machine aids
- ☐ Introduce discipline and training to avoid or mitigate complexity
- ☐ Introduce discipline and training to evaluate the software interface complexity associated with choices in the architecting process
- ☐ Introduce discipline and training to make hard choices to simplify the system under consideration
- ☐ Introduce discipline and training to remove drivers on complexity by removing system features, finding alternate ways to control hazards, or reducing the required degree of flexibility

B.6 Concept Studies

Discover and adjust the system boundary

- ☐ Dividing the system of interest from its environment
- ☐ Define success of the system of interest

- ☐ Check that whatever is inside the system boundary can meet the need, within the range of cost, schedule, and risk considered acceptable
- ☐ Identify what will be considered to be under the control of the project managers and system engineers, to be somewhat amenable to the influence of project managers and system engineers, or to be completely fixed and unchangeable
- ☐ Assess whether influences needed to achieve success are available to the manager and engineer
- ☐ Assess whether accountability for outcomes excludes that which cannot be controlled
- ☐ Identify a combination of physical and nonphysical models that will express the system boundary
- ☐ Assess sufficiency of information in system boundary models to distinguish between what is inside and outside the system from the software viewpoint
- ☐ Define a black-box model of the system in relation to its environment
- ☐ Identify communications channels, networks, server environments, or user platforms
- ☐ Identify physical issues that the software is to manage, such as accuracy of targeting, quality of products emerging from the system, or physical conditions to which the system will be expected to adapt
- ☐ Identify functions relevant to software
- ☐ Assess degree of imprecision and incompleteness of conceptual models against the expectations of schedule, risk, and cost
- ☐ Correct the system boundary when new information becomes available
- ☐ Correct the success criteria when new information becomes available
- ☐ Check for absence of black-box model
- ☐ Check for white-box internal model obscuring external black-box model
- ☐ Assess degree to which modeling is focused on internal vs. external concerns
- ☐ Assess degree of understanding of the system in relation to its environment
- ☐ Propagate differences between hardware and software boundary through the rest of the system design
- ☐ Identify nonmaterial things will be interacting with the environment
- ☐ Check for functions that will interact with the environment
- ☐ Check for Information that will be exchanged with the environment
- ☐ Check for states that will be managed by the system or that exist in the environment
- ☐ Check for behaviors or activities or roles involved in interactions with the environment
- ☐ Check for rules of system or environment operation
- ☐ Check for services or capabilities or ports that the system is expected to provide or to use
- ☐ Check for conditions or desired effects or measures that the system is expected to achieve or cope with
- ☐ Check for conceptual representations of material entities—classes of material objects that will eventually be instantiated in either the environment or the system—that are involved in interactions between the environment and the system
- ☐ Describe specialized software concepts unique to the system under consideration

- ☐ Identify differences between the physical and software boundaries
- ☐ Adjust the system boundary to avoid pathologies
- ☐ Check for pathology where a very high rate of data exchange is needed.
- ☐ Check for pathology where a software interface at a functional boundary where excessive information about the system internals is needed for the environment to perform its functions
- ☐ Check for overly strict alignment of software and hardware boundaries
- ☐ Check for unrecognized high-level software systems
- ☐ Consider using the method of Friedenthal, Moore, and Steiner (ref. 17) to use the Systems Modeling Language (SysML) to precisely define the system context using blocks, ports, and interfaces
- ☐ Consider using SysML to represent nested structures and connectors
- ☐ Compare system contexts from models of the system of interest and its surrounding systems to reveal gaps and inconsistencies in the understanding of the system boundary

Define the problems to be solved—expectations, emergent characteristics

- ☐ Define the expected performance of the system functions
- ☐ Consider expressing expected performance in terms of desired effects
- ☐ Define the expected quality of the solution: how often it can fail; what guarantees are required from the existence or operation of the system
- ☐ Define constraints: when does it need to be ready, what are permissible and impermissible interactions with the environment; what content of the system is permissible and impermissible
- ☐ Define emergent characteristics: what is expected out of the system that does not result from just having a collection of elements
- ☐ Set the stage for the top-level software system design, and defining the software interface requirements
- ☐ Assess understanding of the problem to be solved, expectations, and emergent characteristics from the software viewpoint.

Flesh out boundary information

- ☐ Define who is going to use the system
- ☐ Define temporal aspects of activity expected of the system
- ☐ Define the process, policy, and organizational strategies that contribute to solution of the needs expressed
- ☐ Define how the system interacts with the environment
- ☐ Test conceptual system models against parties in the environment by simulating system activity at the black-box level
- ☐ Assess sufficiency of information regarding operational information from the software viewpoint

Find the system functions

- ☐ Generate functional design descriptions from system models (see, e.g., ref. 5)

Prepare for downstream engineering activity

- ☐ Check for use of nonauthoritative information, inconsistency, or divergence
- ☐ Define the conceptual spaces in which software interfaces are defined
- ☐ Ensure there are means in the modeling environment to distinguish the modeling spaces in which software interfaces exist
- ☐ Define management and engineering processes that are rooted in the properties of the conceptual spaces in which software interfaces exist
- ☐ Define management and engineering processes that recognize, tolerate, and promote management of a nonmaterial world
- ☐ Define management and engineering processes that recognize the existence of a top level software design, even in system whose sub-elements are peers of the system of interest or are legacy systems
- ☐ Clarify who has authority over the top level software design, and accountability
- ☐ Determine which system functions are to be accomplished by hardware and which by software
- ☐ Define the organization and work breakdown structure for software engineering
- ☐ Arrange accountability of element software engineers to the top-level software engineers
- ☐ Prepare a system for tracking technical performance, cost, schedule, risk; include software in tracking
- ☐ Prepare a system for expressing software interfaces, both requirements and design, based on the model
- ☐ Check for excessively fractionated efforts
- ☐ Define terminology and conventions of expression
- ☐ Develop well-formedness rules to enable model-based detection of misapplied concepts
- ☐ Use model-based detection to correct misapplied concepts
- ☐ Produce work breakdown structures from the model
- ☐ Check that expectations of any modeling efforts are tied to project deliverables
- ☐ Assess ability to share model information between software interface engineering and the other disciplines
- ☐ Decide how model information in different disciplines will be integrated
- ☐ Determine model transformations that will be needed in the future

B.7 Concept and Technology Development

Define software requirements on systems outside project control

- ☐ Identify systems that are outside project control because they are part of the environment
- ☐ Identify systems that are outside project control because they are legacy systems to be incorporated into the system-of-interest but are no longer in development
- ☐ Identify systems that are outside project control because they are a new or modified system to be incorporated into the system-of-interest but authority over them belongs to some other project

- ☐ Use models to clarify the expectations that the developers of the system of interest have upon peer systems
- ☐ Use models to ask the question "does it work like this"
- ☐ Use models to answer the question "does it work like this"
- ☐ Design the new system to match legacy systems

Manage software interface requirements and constraints of many types

- ☐ Define software interfaces using each of the conceptual spaces inhabited by the interface
- ☐ Define the conceptual temporal software interface characteristics
- ☐ Decompose the conceptual functions to a level that can be allocated to the sub-elements
- ☐ Define the temporal characteristics of the decomposed functions
- ☐ Assess the sufficiency of information regarding what is expected from the sub-elements from the software viewpoint

Plan to overcome software interface technological limitations

- ☐ Develop models that contain sufficient information to manage technological limitations

Select external software interface architectures

- ☐ Select external software interface architectures

Find the system functions again

- ☐ Generate functional design descriptions from system models (see, e.g., ref. 5), this time taking into account relationships between requirements

Partition the system and software

- ☐ Assess robustness of software interfaces to changes in external conditions (including potential future changes of interface architecture)
- ☐ Assess sensitivity to, and management of faults by, software interfaces
- ☐ Assess ability of software interfaces to realize the desired emergent characteristics of the system as a whole
- ☐ Assess whether the assumptions of boundary and interior regions are realized
- ☐ Consider adding design features to realize the assumption of boundary and interior regions as an emergent characteristic
- ☐ Check for sneak paths between conceptual spaces that create unintended interactions between elements of the software partitions
- ☐ Develop well-formedness rules to enable model-based detection of illusions of separation

Explore emergent software characteristics

- ☐ Identify derived functions that arise out of the expectations at the system level
- ☐ Trace using a requirements model, to ensure that the derived functions are engineered into the functions of the system

Define models of software interfaces

- ☐ Use models of interfaces to convey requirements

- ☐ Derive implementations of interfaces from models, eliminating errors that can arise from translation into and out of documents containing natural language
- ☐ Construct early, operable models of interfaces
- ☐ Confirm compatibility of software interfaces locally by exchanging operable models between the software developers

Define responsibility for software interface definition

- ☐ Consider using a single model of each interface, with shared governance of that model, to define interfaces

Describe software interfaces from the models

- ☐ Derive software interface descriptions from the software that implements the interfaces

Plan software interface content

- ☐ Plan software interface content, to address all the relevant aspects of the conceptual spaces in which a particular interface exists
- ☐ Consider using a selectable content standard based on the models defining the conceptual spaces in which an interface exists
- ☐ Make plans for defining the space-specific details of each interface
- ☐ Make plans for reference models for protocols or commonality standards, to be used later in detailed software interface specification

Track aggregated software interface characteristics

- ☐ Track aggregated software interface characteristics

Plan model exchanges

- ☐ Make plans for the ongoing exchange of models for requirements or description purposes

B.8 Preliminary Design and Technology Completion

Define the system context

- ☐ Allocate functionality, persistent data and control information among the logical and physical components of the system
- ☐ Allocate system interfaces to physical components (hardware or software)
- ☐ Describe things beyond the scope of software implementation; that is, describe the problem domain, not the solution (software) domain
- ☐ Define the physical parts of the system—external systems, users, interfaces, communication channels, etc.
- ☐ Define the informational context of the system—what artifacts exist, are produced, and are inputs or outputs and how these elements relate to each other
- ☐ Confirm that all the data in the context model will exist irrespective of what software solution is developed
- ☐ Confirm that the data in the context model is such that if it changes, then the understanding of the system problem has changed
- ☐ Confirm that there are not too many implementation details in the use cases

Develop system functional architecture

- ☐ Decompose the system into logical components including external interface components, application components and infrastructure components
- ☐ Use sequence or activity diagrams to model software components and also system-level behaviors
- ☐ Evaluate the degree of non-overlapping and collaborating components
- ☐ Confirm that modeling is done on at least at two different levels: the analysis level and the design level
- ☐ Component diagrams can be used to model the logical architecture of a system

Design system physical architecture

- ☐ Allocate logical components to physical components
- ☐ Check whether allocation decisions answer performance, reliability, security and other system issues that impact interfaces
- ☐ Derive interface requirements from black-box interactions or black-box system requirements
- ☐ Use deployment diagrams to model the physical architecture of a system by first showing nodes and devices and connecting them using communication paths
- ☐ Model the components that run in each of the physical nodes
- ☐ Model the applications that run on the different nodes and the components that make up the applications (for example, application artifacts may be wired to nodes through the 'deploy' connector, and wired to components through the 'manifest' connector)
- ☐ Consider using artifacts to represent the external application

Find the last of the system interface-related functions and characteristics

- ☐ Design emergent software characteristics
- ☐ Identify repeated patterns of interactions (interfaces are essentially drawn from interactions with an outside system or environment)
- ☐ Define interaction use cases
- ☐ Define software system functions such as shut down, start-up and fault management functions (FDIR)
- ☐ Assess performance characteristics of these functions affected by the data provided across interfaces
- ☐ Assess interface characteristics:
- ☐ 1) Priority assigned to the interface by the interfacing entities
- ☐ 2) Type of interface (e.g., real-time data transfer, storage and retrieval of data, etc.)
- ☐ 3) Characteristics of the individual data elements that are provided across the interface, such as name, data type, size, format, unit of measure, rage, accuracy, priority, timing, frequency, source of data, constraints (security, privacy, etc.)
- ☐ 4) Characteristics of the data aggregates (messages files, records, etc.) provided across the interfaces, such as name formats, data structure, sorting or access characteristics, priority, timing, frequency, volume, sequencing, constraints, etc.

- ☐ 5) Characteristics of the communication method used by the interfacing entities, such as name, comm link, message formatting, flow control, data transfer rate, routing, addressing, naming conventions, encrypting, authentication, etc.
- ☐ 6) Characteristics of the protocol layer, such as name, priority, packetizing, error control and recovery, synchronization, connection information, reporting features, etc.
- ☐ Obtain interface information from inputs and outputs of scenarios, use case, and system state machines associated with scenarios and system behaviors
- ☐ Consider using the concept of interaction boundary, and formal gates where information that leaves or enters the boundary gets formally defined

Propagate system flows to system boundaries

- ☐ Decompose interactions, identifying nested interactions
- ☐ Propagate internal flows through the system until they reach the system boundaries
- ☐ Capture propagated internal flows in the formal software interfaces between systems
- ☐ Define the interactions between software and system interfaces, hardware interfaces and other boundaries
- ☐ Use ports are used to represent an interaction point at a boundary
- ☐ Flow ports describe flow in and through a boundary point
- ☐ Service ports describe what services are required or provided at the boundary
- ☐ Describe software items that flow
- ☐ Check for new interface needs; typical issues are incompleteness, overdetermination, incompatibility, transecting across data complexity (requiring knowing too much information at the interface), transecting across excessive data rates

Describe software data flows

- ☐ Use SysML "items" to model software flows
- ☐ Identify types of items in software flows; items may be blocks, value types, or signals
- ☐ Describe software behavior
- ☐ Use service ports to model behavior across software components
- ☐ Check for multiple sources for software interface requirements and constraints (system decomposition (interface allocation), interface standards and protocols)
- ☐ Identify model Items in a library containing "standard" interface items and/or special ones
- ☐ Use libraries of interfaces to produce interface documents

Define software interface standards

- ☐ Check for needs for standards; interface, consistent representation and styles across modeling activity
- ☐ Check for gaps or inconsistencies of interfaces modeling standards and conventions
- ☐ Define an interface taxonomy to specify physical and logical classification of interfaces
- ☐ Consider whether tool-specific notations are easily compatible with standards

Select internal software interface architectures

- ☐ Identify cooperating entities in separate systems
- ☐ Check for compatibility between two ports (specifications, types, direction, etc.)

- ☐ Consider integration brokers at the software interfaces
- ☐ Check for compatibility of the interfaces of standard ports by verifying the compatibility of the operations on the interfaces (see reference 9 for a formal list of incompatibilities)

Integration brokers at the software interfaces

- ☐ Consider properties of communicating brokers
- ☐ Determine how to interface with COTS code that is normally provided for communication protocols
- ☐ Consider using software interfaces at locations other than physical connections

Identify layers of compatibility

- ☐ Identify layers of compatibility (see section B-3.3.9)

Allocate functionality to software or subsystems

- ☐ Allocate functionality to software or subsystems
- ☐ Check for data that flows beyond the I or O from and to the nearest components
- ☐ Confirm that data flows through the entire system from inception to final dispositions

Allocate data and control to software interfaces

- ☐ Allocate to system interfaces
- ☐ Check functional completeness
- ☐ Balance interfaces on the functions
- ☐ Balance interfaces on end-to-end flows
- ☐ Test end-to-end flows

Derive software interface designs to meet system emergent characteristics

- ☐ Check for new hardware interfaces
- ☐ Check for new aggregation or analysis of the interface and software data
- ☐ Search for unintended interactions between items that seem separate at another level.

Derive software interface designs to meet interface requirements and constraints

- ☐ Confirm satisfaction of explicit interface requirements on software (commands, files, status parameters, telemetry, protocols, communication standards, network).

Aggregate software-internal sources and sinks

- ☐ Aggregate software-internal sources and sinks

Define software interfaces between models

- ☐ Define software interfaces between models

Detect software interface (in)compatibility

- ☐ Detect software interface (in)compatibility (see references 8 and 9)

Discover preliminary unintended consequences

- ☐ Ensure model content and applicability is suitable to discover unintended consequences
- ☐ Create models of sources and sinks behavior
- ☐ Create models of aggregated and chained sources that describe the flow of information through a system

Iterate the SoS software design

- ☐ Check the software design using executable models, run themes and variations on key interface scenarios

Verify preliminary software design

- ☐ Use executable behavioral models to verify software designs
- ☐ Execute mission timelines with models
- ☐ Identify key interface questions that must be answered at a successful PDR
- ☐ Plan how to attain the answers using models and simulation techniques
- ☐ Plan for interface document generation
- ☐ Consider approach to verification: completeness or exhaustive verification vs. spot checking key interfaces and requirements

Design software verification and validation system

- ☐ Use model checking tools and other automated verification and validation systems to check life cycle activities: requirements, design, and implementation
- ☐ Maintain interface baseline through SysML IBDs and message model elements.

B.9 Final Design and Coding

Design detailed software interfaces

- ☐ Enumerate data
- ☐ Refine interface behaviors
- ☐ Select specific parameters of the interface architecture
- ☐ Capture detailed software interfaces as refinements to the interface models
- ☐ Exercise detailed interface models to verify compatibility between the interfacing elements

Manage changes

- ☐ Capture dependencies on model characteristics in a machine-readable form
- ☐ Prepare an automated workflow system
- ☐ Identify dependencies a priori from the software architecture
- ☐ Discover dependencies from a record of access to model information; parties who access model information are likely to have a concern if it changes; this can be confirmed in an automated dialog with the accessing party
- ☐ Automatically notify parties of changes in model information

Discover refined unintended consequences

- ☐ Exercise collections of detailed interface models in the context of simulated missions

Define software manufacturing process

- ☐ Define software manufacturing process in a model
- ☐ Use the manufacturing process model to automate workflow
- ☐ Use the manufacturing process model to audit compliance and completeness of the process as executed

Determine readiness for software coding

- ☐ Check completeness and consistency of the requirements automatically by examining the models conveying the requirements
- ☐ Consider postponing the final software coding based on incompleteness and inconsistencies
- ☐ Identify risks associated with partial codings
- ☐ Express standards for readiness in a model
- ☐ Automatically evaluate readiness using the model
- ☐ Assess quality of decision-making regarding progress on software development

Code the software

- ☐ Automatically generate code from structure, data, and behavioral models used to express interface requirements
- ☐ Generate documentation of as-built interface directly from code

Discipline to software interface definitions

- ☐ Use models to enhance discipline to software interface definitions
- ☐ Generate detailed descriptions are generated automatically from code, then discipline to the definition arises automatically
- ☐ Generate code or detailed descriptions manually, then verify the actual interface against the model in the interface requirements; check for evidence of deviations that can then be used to justify corrections
- ☐ Consider interface verification as soon as stub software routines are defined to the extent needed to mimic the interface

Implement software verification and validation system

- ☐ Capture the verification and validation process in a model
- ☐ Use the model to automate workflow
- ☐ Use the model to audit compliance and completeness of the process as executed
- ☐ Check for software interface structure, content, and behavior
- ☐ Begin implementation of the software verification and validation system during the software interface design process
- ☐ Consider doing some verification automatically based on the models of the software interfaces and architecture
- ☐ Prepare actual hardware on which to verify or validate the software
- ☐ Consider doing some verification automatically on actual hardware

Apply software interface standards

- ☐ Consider using reference models to incorporate large amounts of detailed interface information from references, protocols, or commonality standards
- ☐ Arrange for every interfacing party to incorporate the reference models into their software development
- ☐ Check for incompatibilities that arise from a multitude of locally developed models for the references
- ☐ Check for duplicative costs developing a multitude of local models

Verify final software interface design

- Test software interfaces automatically against the model in the interface description

Verify using virtual software implementations

- Consider verifying the understanding of the software interface before investing in software implementation
- Consider placing interface models into a discrete-event simulation engine, which then can be operated to demonstrate the expected behavior of the software

B.10 System Assembly

- Define a model of the software verification and validation system
- Assemble software automatically based on the model
- Test conformance to software interface design
- Test software interfaces automatically against the model in the interface description
- Determine readiness for integration and test
- Check completeness and consistency of the coding by comparing the models conveying the requirements with the models describing the as-built software interfaces
- Express standards for readiness in a model
- Automatically evaluate readiness using the model

B.11 Integration and Test

Validate using virtual software integration and test

- Place the actual software into a collection of virtual machines, which then can be operated to demonstrate the expected behavior of the entire system
- Define the structure of the software system in the requirements and the as-built software model

Integrate software components

- Automatically integrate software based on the specified structure

Integrate software verification and validation system

- Define the structure of the software verification and validation system in the requirements and the as-built software model
- Automatically integrate the software verification and validation system based on the specified structure

Test and verify software design

- Generate the software test and verification design automatically from the higher level model in the software system architecture
- Derive test cases and expected results from the architecture
- Where some verifications should be done by a combination of test, analysis, inspection, or demonstration, automate the workflow of the assessment process based on the software architecture model

Execute software tests and verifications

- When verification of software interfaces by test and demonstration is required, do this automatically by supplying inputs based on the model and comparing outputs with the expectation from the model
- When human or hardware interaction is required, automatically supply prompts to humans on what to do and what to expect in response
- When verification by analysis or inspection cannot be done automatically, automatically generate the analysis and inspection requirements from the system model

Capture evidence

- Specify in a model which evidence is required, where and when it is to be obtained, who is to obtain it, and who is to judge whether it is satisfactory
- When the evidence is machine-readable, send it automatically to its intended destination
- Schedule tests and participants automatically based on the model
- When evidence can only be collected by witnesses, supply witnesses with prompts on what to do and what to expect in response

Validate software

- Exercise the as-built software system according to design reference missions expressed in the model of stakeholder expectations

Determine system acceptability

- Report aggregated metrics specified by the system model, and traced to metrics from lower-level elements

Discover final unintended consequences

- Exercise the entire as-built system in realistic operations to reveal any unintended consequences

B.12 Launch

Configuring the software system

- Configure the software system using the system model
- Set up build profiles based on system composition and interface connectivity specified in the system model
- Set software interface parameters as defined in the system model
- Where hand input is required, export configuration instructions directly from the system model

Software readiness for launch

- Use models to compute readiness metrics for software interfaces, including verification status, bug reporting, and software process status
- Use parametrics in the system model to assess readiness of system functions, traced to verification evidence for components

Virtual Missions

- Use executable models to simulate software interactions with real systems to conduct virtual missions
- Use executable models for training and final mission readiness testing

B.13 Operations and Sustainment

Training

- Use executable models for training, similar to virtual missions but more focused to specific system-with-operator-in-the-loop interactions
- Use information in the model to develop training materials based on system structure, behavior, and interface characteristics

Software system diagnosis

- Compare behavior of executable models to actual system behavior to either detect anomalous software operation or to adjust models to more accurately reflect actual system operation
- When a hypothesis is formed regarding the reasons for anomalous operation, test the corrections with the executable model to mitigate risk of unintended consequences
- Use definition information in the model to look up the meaning of system control, monitoring, and performance data as it is observed in real time

Acquisition of software data

- Use models to compute performance metrics for software interfaces, such as loading and response time, compute margin, etc.
- Connect the model to real time data acquisition so that derived quantities can be calculated relative to actual data

Software system upgrades

- Compare behavior of executable models with and without upgraded software to discern the aggregated consequences of upgrading software on overall system performance
- Mitigate risk of unintended consequences from software upgrades

B.14 Closeout

Analysis of software data

- Use models to compute higher level performance metrics for software interfaces, such as system effectiveness, e.g., monitoring of operator workloads, consumption of system consumables, and system failure history

Software reuse

- Identify final software structure and interfaces for inheritance on later projects
- Evaluate software interfaces for sources of future architectural incompatibility
- Capture final, as-modified-during-operation software interfaces for inheritance on later projects

Software disposal

- Retain models, executable models, simulations as permanent records final software structure and interfaces for inheritance on later projects

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
01- 08 - 2013	Technical Memorandim	May 2011 - July 2013

4. TITLE AND SUBTITLE
Assess/Mitigate Risk through the Use of Computer-Aided Software Engineering (CASE) Tools

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Aguilar, Michael L.

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER
869021.05.07.07.27

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
NASA Langley Research Center
Hampton, VA 23681-2199

8. PERFORMING ORGANIZATION REPORT NUMBER
L-20314 NESC-RP-10-00609

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Washington, DC 20546-0001

10. SPONSOR/MONITOR'S ACRONYM(S)
NASA

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
NASA/TM-2013-218031

12. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified - Unlimited
Subject Category 61 Computer Programming and Software
Availability: NASA CASI (443) 757-5802

13. SUPPLEMENTARY NOTES

14. ABSTRACT

The NASA Engineering and Safety Center (NESC) was requested to perform an independent assessment of the mitigation of the Constellation Program (CxP) Risk 4421 through the use of computer-aided software engineering (CASE) tools. With the cancellation of the CxP, the assessment goals were modified to capture lessons learned and best practices in the use of CASE tools. The assessment goal was to prepare the next program for the use of these CASE tools. The outcome of the assessment is contained in this document.

15. SUBJECT TERMS
Constellation Program; Computer-Aided Software Engineering; NASA Engineering and Safety Center; Model-based systems engineering; Interface Control Document; Computer-aided design

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			STI Help Desk (email: help@sti.nasa.gov)
					19b. TELEPHONE NUMBER *(Include area code)*
U	U	U	UU	188	(443) 757-5802