# NASA's Advanced Multimission Operations System: A Case Study in Software Architecture Evolution

Jeffrey M. Barnes*
Institute for Software Research
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, Pennsylvania 15213
jmbarnes@cs.cmu.edu

## ABSTRACT

Virtually all software systems of significant size and longevity eventually undergo changes to their basic architectural structure. Such changes may be prompted by new feature requests, new quality attribute requirements, changing technology, or other reasons. Whatever the cause, software architecture evolution is commonplace in real-world software projects. However, research in this area has suffered from problems of validation; previous work has tended to make heavy use of toy examples and hypothetical scenarios and has not been well supported by real-world examples. To help address this problem, this paper presents a case study of an ongoing effort at the Jet Propulsion Laboratory to rearchitect the Advanced Multimission Operations System used to operate NASA's deep-space and astrophysics missions. Based on examination of project documents and interviews with project personnel, I describe the goals and approach of this evolution effort, then demonstrate how approaches and formal methods from previous research in architecture evolution may be applied to this evolution while using languages and tools already in place at the Jet Propulsion Laboratory.

## Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*restructuring, reverse engineering, and reengineering*; D.2.11 [**Software Engineering**]: Software Architectures

## General Terms

Design

## Keywords

Software evolution, case study

*The author was a 2011 summer intern at the Jet Propulsion Laboratory, California Institute of Technology.


*QoSA'12,* June 25–28, 2012, Bertinoro, Italy.


## 1. INTRODUCTION

Software architecture is today widely accepted as an essential means of designing software systems effectively. However, one topic that existing approaches to software architecture do not address well is *software architecture evolution.* Software architecture evolution is a phenomenon that occurs in virtually all software systems of significant size and longevity. As a software system ages, it often needs to be structurally redesigned to support new features, incorporate new technologies, adapt to changing market conditions, or meet new architectural quality requirements. In addition, many systems over the years tend to accrue a patchwork of architectural workarounds, makeshift adapters, and other degradations in architectural quality, requiring some sort of architectural overhaul to address.

At present, however, software architects have few tools to help them plan and carry out these kinds of architecture evolution. Currently, such evolution is usually planned and carried out ad hoc, with only informal planning. However, informal planning has limitations, and large-scale software evolutions are complicated by numerous uncertainties and conflicting concerns that make such planning difficult. Many of today's large-scale software evolutions are plagued by unnecessary backtracking, rollbacks, and other symptoms of inadequate or inaccurate planning. Some of these problems might be alleviated if software architects had better guidance when planning evolutions, in the form of tools and models for architecture evolution planning.

While there is a sizeable body of research literature on software maintenance and evolution generally, little work has been devoted to this common problem of architecture evolution. In previous work [13], we sought to address this gap. There, we discussed the phenomenon of software architecture evolution and presented a model to support architects in planning such evolutions. Our approach is based on capturing architectural expertise about classes of evolutions and developing tools to facilitate reuse of this expertise.

One significant obstacle in our research thus far, as well as that of other researchers working in the area, has been the challenge of validation. Though we have made considerable headway in developing a theory of architecture evolution, finding real-world cases on which to test our model has been a challenge. Most work in this area has been based on toy examples and artificial evolutions in laboratory conditions.

This paper presents the result of a collaboration between Carnegie Mellon University and NASA's Jet Propulsion Lab-

oratory (JPL) with the aim of understanding the software architecture evolution that occurs at JPL and assessing the degree to which our approach to architecture evolution can be helpful in understanding real-world evolution problems.

The main goals of this work were to (1) understand a real-world software architecture evolution problem in its natural context, (2) assess the usefulness of our framework for software architecture evolution in helping to plan evolutions and reason about trade-offs, and (3) assess the ease of implementing our approach to software architecture evolution with off-the-shelf languages and tools. Section 6 will revisit these goals.

The rest of the paper is organized as follows. Section 2 describes our approach to software architecture evolution in greater detail. Section 3 describes the design of this case study. Section 4 begins with a detailed description of the evolution examined by this case study and goes on to describe how our approach to architecture evolution was applied to this case. Section 5 surveys related empirical research in architecture evolution. Section 6 discusses the broader conclusions that may be drawn from this case study and the research contribution that the case study makes.

# 2. BACKGROUND

## 2.1 Software Architecture

Software architecture is the subdiscipline of software engineering that pertains to the overall structure of a software system. Software architects represent software systems in terms of the high-level elements from which they are made. The most important kinds of architectural elements are *components* (the computational elements and data stores of a system) and *connectors* (interaction pathways among components) [5]. At a basic level, a software architecture can be thought of as an arrangement of components and connectors. In practice, architectural descriptions may be made more complex by the addition of other kinds of architectural elements, the annotation of architectural elements with properties to facilitate analysis, and other intricacies.

## 2.2 Software Architecture Evolution

The problem of understanding software architecture evolution is just beginning to be explored. In previous work, we developed an approach for understanding and modeling software architecture evolution, supported by automatable formal methods [13]. This section provides a brief summary of this approach; for further details, see the previous paper.

We assume that there is a known start state (the current state of the architecture) and a known end state (the goal of the evolution). In practice, this assumption may not hold. However, other research areas address the problems of determining the architecture of an existing system and designing an architecture for a future system (architecture reconstruction and architectural design, respectively), so we do not address them. Instead, we are concerned with how to get from the current state to the target state.

We start by contemplating the set of potential *intermediate states* between the initial architecture and the target architecture—the transitional states that the system may assume as it evolves from its initial form to its target architecture. We represent all the intermediate states, together with the initial and target states, as nodes in a directed graph. We then draw an edge from node $a$ to node $b$ if there is an evolutionary transition from state $a$ to state $b$. In addition, we allow nodes and edges to be annotated with an extensible set of architectural properties that further characterize the evolution. These properties support analysis of the evolution.

This conceptual setup is fairly simple, but it accomplishes a few things. First, it allows us to see the different ways that a system can evolve. In particular, it allows us to consider the set of possible *evolution paths*—complete routes from the initial state to the target state—and consider trade-offs among them. We can also visualize things like release points and milestones by demarcating them with node properties.

This setup is also amenable to various kinds of analysis. One of the simplest such analyses is the analysis of which evolution paths are possible or legal. One of the basic elements of our model of architecture evolution, therefore, is the notion of a *path constraint*, an analysis indicating which paths are legal with respect to some rule about the evolution domain. Formally, a path constraint is a predicate over evolution paths; for each path, a constraint either allows it or forbids it. An example of a constraint is: "Once a component is migrated to a data center, it must remain there for the rest of the evolution." For any given evolution path, this constraint will either hold or fail to hold. Another important class of analyses is *path evaluation functions*. While a path constraint provides a judgment about the *legality* of a path, a path evaluation function instead provides an assessment of the *quality* of a path, for example by estimating its duration or cost. Evaluation functions can be used to support selection of ideal paths (i.e., paths that are optimal with respect to the qualities important to the evolution at hand).

These concepts are fairly simple, but there is a substantial formal framework supporting them. To formalize path constraints, for example, we have developed a formal language based on linear temporal logic. The use of formal methods has several advantages, the most important of which are precision (by using a formal approach, we minimize ambiguity, which helps to pin down what project stakeholders mean when they talk about the project) and automation (a formal approach allows us to develop tools to make it easier to plan and analyze the evolution).

# 3. CASE STUDY DESIGN

A case study is a particularly suitable methodology for studying software architecture evolution. In general, the primary use of a case study is that it provides a way of studying a phenomenon in its real-world context. Case studies are most useful in domains where more robust empirical methods are impractical. Software architecture evolution is such a domain; instances of industrial-scale architecture evolution often take months or years to complete and entail the expenditure of substantial resources, making controlled experiments infeasible.

Of course, case studies have significant drawbacks and limitations as well. In particular, it is more difficult to generalize the results of a case study than of a study based on quantitative methods. Section 6 will discuss the issue of generalizability with respect to this case study.

## 3.1 Research Questions

The case study was intended to address questions such as:

1. How common is software architecture evolution in a real-world software organization?

2. How do practicing software architects think about and deal with the realities of architecture evolution?

3. Do practicing software architects recognize architecture evolution as a problem?

4. What tools do software architects use to deal with architecture evolution? Do they find them adequate?

5. Can the approach to architecture evolution that we have developed in our research help architects to plan evolution?

6. How easily can our approach be put into practice?

## 3.2 Subject Selection

A number of factors made JPL a particularly suitable environment for exploring these questions. First, JPL is a much older organization than some other kinds of software organizations one might study (e.g., Internet companies or open-source projects) and has some very old software systems, making it a potentially rich vein to mine for evolution instances. NASA missions can last decades, and software systems must evolve to support them continuously. Of course, NASA is by no means unique in having to maintain very old software systems—similar problems are commonplace in industries such as telecommunications and banking—but the longevity and complexity of JPL's software systems, along with the accessibility of information on them, made JPL a suitable venue for exploring software architecture evolution in practice.

In addition, based on preliminary discussion with JPL personnel, we were aware of a major, ongoing architecture evolution that seemed likely to provide a suitable case: the evolution of the Advanced Multimission Operations System (AMMOS), the ground software system used for JPL's deep-space and astrophysics missions [16]. Section 4 will discuss this evolution in detail.

Finally, there were pragmatic considerations. Industrial case studies are often subject to onerous confidentiality agreements that impose serious restrictions on the information that researchers can share. JPL, by contrast, is a government laboratory operated by a university and so is generally friendly to dissemination of research.

## 3.3 Case Study Structure

In this case study, I spent ten weeks in the ground systems engineering section at JPL. The case study proceeded in two phases.

During the first few weeks, the focus was gathering information. I spent most of this phase reviewing project documents and speaking with personnel to familiarize myself with the elements of AMMOS and the plan for evolving them. This phase was *descriptive* and *exploratory* in character; the main goal was to understand the context of architecture evolution at JPL and learn how engineers think about and deal with architecture evolution. The first phase chiefly addressed research questions 1–4 from the list above. A secondary goal was to prepare for the second phase of the case study by selecting a specific evolution to study.

In the second phase of the case study, I applied our architecture evolution research to model the selected evolution. This phase was *evaluative* in character; the goal was to assess how easily and how usefully our approach could be applied in a real-world setting. It addressed research questions 5–6.

## 3.4 Case Selection

During the first phase of the case study, an number of specific evolution instances were under consideration for detailed study in phase 2. I worked with my project contacts at JPL to select the most appropriate choice. Among the options were past evolutions (those that had been finished and whose outcome was known); current evolutions (those which were ongoing); and future evolutions (those under consideration for the future). We ultimately selected an evolution that was in progress. Picking a current evolution had the advantage of being of greatest relevance to JPL. Another advantage was that there were ample resources for learning about the evolution; it was easy to find project personnel who could share accurate, timely information about evolution plans. If we had selected a past evolution, documentation would have been harder to find, and few personnel would have been familiar enough with the evolution to provide useful information. If we had selected a future evolution for which few firm plans had been made, we would have had to engage in substantial speculation to construct an evolution graph.

Another important choice was the scope of the evolution, in terms of both time (i.e., a long evolution versus a short one) and breadth (i.e., the evolution of a small subsystem versus the evolution of a large chunk of AMMOS). The evolution we selected was relatively small in both dimensions; that is, the second phase of the case study focused on a modestly sized subsystem of AMMOS and modeled evolution plans only a couple of years into the future. Though we could have picked a larger scope, time constraints would have made it difficult to gather sufficient information to produce a useful model—one that was more than a superficial overview. A more narrowly scoped example than the one we selected, on the other hand, would have shown changes too minor to be interesting.

The specific evolution instance we selected had several other advantages. First, it had an explicit initial software architecture, and the target architecture was also fairly well understood. Second, it presented interesting trade-offs and questions that might be usefully addressed by architecture evolution analysis. Third, I had good access to staff who were familiar with the system and the evolution, who could provide architectural information beyond that available in documentation.

# 4. RESULTS

## 4.1 Case Description

One reason that software evolution happens often at JPL is that not only must software support long missions, but also software often must support many different missions. Each JPL mission has plenty of custom software written for it, but most missions also make use of *multimission software*—software shared among several missions. JPL takes a sort of product line approach to multimission software; it develops software for multimission use, then adapts it for each mission. As new missions make use of a multimission platform, the platform must evolve to support the new capabilities and qualities that the new missions require. Over a long period of time, a multimission system can change drastically, ultimately to the point where it bears little resemblance to its ancestral form.

The best example of such a multimission system at JPL, and the one examined in the case study, is the Advanced

Multimission Operations System (AMMOS). AMMOS is the ground software system used for JPL's deep-space and astrophysics missions [16]. It was developed beginning in 1985, with the goal of providing a common platform to allow mission operators to manage ground systems at lower cost than would be possible by building mission-specific tools, without compromising reliability or performance [14]. The system has been used for many prominent NASA missions and continues to be used today.

Architecturally, AMMOS is a *system of systems*; though it functions as a coherent whole with a unified purpose, it is composed of disparate elements, each with its own engineers, users, and architectural style. Among the systems making up AMMOS are elements responsible for uplink and downlink of spacecraft telemetry, for planning command sequences, for processing spacecraft telemetry, for navigation, and so on [16].

AMMOS has served JPL well for many missions, but it is an aging system, and its architectural limitations are becoming apparent [23]. The architecture is resistant to evolution and expensive to maintain. The current system suffers from architectural inconsistencies and redundancies and lacks a coherent overarching architecture. Requirements changes often necessitate modifications that span many subsystems, and the system relies on large amounts of "glue" code—adapters and bridges connecting different parts of the system in an ad hoc way that makes maintenance difficult.

Now, ongoing architecture modernization efforts aim to address these quality problems by rearchitecting AMMOS in a way that makes use of modern architectural styles and patterns [23, 27]. This will allow easier, less expensive maintenance and evolution of AMMOS in the future and also facilitate easier customization of AMMOS for individual missions. The goal is to develop a modern deep-space information systems architecture that supports the qualities of composability, interoperability, and architectural consistency.

## 4.2 Evolution Details

The second phase of the case study focused specifically on the AMMOS element responsible for mission control, data management and accountability, and spacecraft analysis (MDAS). The MDAS element has a number of responsibilities, but one of the most important is to process, store, and display telemetry and other mission data from deep-space operations. Prior to the Mars Science Laboratory (MSL) mission, this responsibility was fulfilled by an assortment of different subsystems: the Data Monitor and Display assembly; the Tracking, Telemetry, and Command system; and a number of others [17]. For the MSL mission, a new system was developed to supplant this complex of systems: the Mission Data Processing and Control System (MPCS).

MPCS was originally developed as a testing platform modeled after the ground data systems for the Mars Exploration Rovers; later it was promoted to support operations for MSL [8]. Engineers are now adapting and refining the architecture of MPCS for multimission use.

MPCS has an event-driven message bus architecture. All the major components of the system communicate via a Java Message Service message bus [8]. This promotes loose coupling of software components without compromising reliability. Components can attach to or detach from the message bus freely (by subscribing to or publishing the appropriate kind of event), provided that they adhere to application protocols and do not violate architectural constraints, allowing for plug-and-play reconfiguration of the system. The components are Java-based and platform-independent; the interfaces by which they communicate are based on XML.

This event-driven, bus-mediated architecture gives MPCS architectural flexibility. There is not really any one "MPCS architecture;" rather, MPCS can be configured in different ways to achieve different goals. At its most flexible, MPCS is a loose confederation of tools rather than a cohesive system with a fixed design. However, MPCS does have a rather stable infrastructure of core components that are usually connected in a well-defined way, so we can generally treat MPCS as a system with a stable platform and fixed variation points.

An important example of the architectural variability of MPCS is that it is deployed with different configurations in different environments. MPCS is used in several environments—not only mission operations, but also flight software development; system integration; and assembly, test, and launch operations (ATLO)—and there are significant differences in architectural configuration among them [8]. For flight software development, for example, MPCS can be used to issue commands to the flight software under development; in operations, commanding features are delegated to a different system called CMD, which is external to AMMOS.

The most important components of MPCS are:

- The aforementioned **message bus**.
- The telemetry processing subsystem of MPCS, called **chill_down** (*chill* is a code name for MPCS [29], and *down* is for *downlink*). This component takes as input an unprocessed telemetry stream from a spacecraft (or other telemetry source, such as a simulation environment), performs frame synchronization and packet extraction, and processes packets to produce event verification records and data products [2].
- The commanding component of MPCS, called **chill_up** (*up* for *uplink*) [7, 9]. This component transmits commands to the flight software (or simulation environment). Currently chill_up is used only in the flight software development and ATLO environments, not in operations.
- The **MPCS database**, a MySQL database used for storing telemetry and other information, such as logs and commanding data [9]. This database is queried by a number of analysis components.
- The monitoring interface, **chill_monitor**, used for real-time display of telemetry [7, 9]. There are generally many instances of chill_monitor for one MPCS instance, as many mission operators may be monitoring telemetry at once.
- Various **MPCS query** components with names like *chill_get_frames* and *chill_get_packets*, which output data from the database in a standard format [9].

Together, these components effectively form a standard workflow. Commands are issued by chill_up and conveyed to the flight software (or simulation environment), which carries out the commands; the flight software produces telemetry, which is processed by chill_down. The chill_down component stores the processed telemetry to the MPCS database (where

it is queried by the MPCS query components) and transmits messages about the processed telemetry to the message bus (where it is displayed by chill_monitor). Although MPCS can be configured in many different ways, this workflow describes the way MPCS is used most of the time.

This architecture is expected to serve adequately for the MSL mission. However, as MPCS is developed for reuse in future missions, engineers must evolve the system to improve qualities such as performance and usability, support additional capabilities, and better integrate with other ground data systems. The case study focused on two proposed features of MPCS that project architects hoped to introduce in future versions: *integrated commanding* (ICMD) and *timelines*.

ICMD is motivated by the NASA principle "test like you fly." That is, NASA aims to make testing environments as similar as possible to actual spaceflight operations. As we have seen, a salient architectural characteristic of MPCS is that it takes different forms in different environments. There are important architectural differences between the testing environment (ATLO) and the spaceflight operations environment. The ICMD effort aims to bring the operations environment more in line with the ATLO environment.

The main difference between the testing and operations configurations of MPCS is commanding. In ATLO, the chill_up component of MPCS is responsible for issuing commands to the spacecraft. In operations, the responsibility of issuing commands is excised from MPCS entirely; instead, the CMD system is responsible for commanding. ICMD will change the operations environment to look more like ATLO; the chill_up component of MPCS will issue commands in all environments.

Timelines are a new data structure proposed for storing streams of time-oriented data throughout AMMOS. A "timeline" is exactly that: a linear sequence of events with associated times, in chronological order. Many of the kinds of data that JPL handles on a day-to-day basis fit naturally into this model: telemetry, command sequences, and others. The timeline proposal defines specific formats for storage and transmission of timelines and describes the architectural infrastructure necessary to support them. Timelines are expected to be useful for many purposes, but one of the most important is comparison of actual telemetry with expected telemetry. Mission operators need this capability all the time (comparing an observation with a theoretical prediction is one of the most basic requirements in science), but comparing telemetry streams is a manual, laborious operation today. Supporting timelines will require substantial architectural infrastructure. Although the basic idea is not complex, there are stringent performance requirements; processing timelines must be very fast. Thus, for example, a special-purpose timeline database is planned, which will be designed specifically for efficient storage and retrieval of timelines.

The introduction of timelines will have ramifications for many AMMOS elements, including MPCS. Currently MPCS stores telemetry information in a MySQL database. Ultimately this database will be rendered obsolete by the introduction of timelines. After timelines are integrated into MPCS, telemetry will be stored in an AMMOS-wide timeline management system, and the MySQL database will eventually be retired. Other parts of MPCS will also be affected by the introduction of timelines; for example, the subsystem for mission planning and sequencing is likely to see changes as well.

## 4.3 Representing Architectural Snapshots in SysML

Our approach to architecture evolution is based on explicit representation of the set of intermediate evolution states—in other words, snapshots of the transitional states that the system will pass through on the way from the initial architecture to the target architecture (as well as snapshots of the initial and final states themselves, of course). Representations of intermediate architectures are thus one of the primary artifacts involved in planning architecture evolution.

In previous work, we modeled architecture evolution using research languages and tools. One of the aims of this case study was to evaluate the practicality of adapting our approach to off-the-shelf languages and tools like those used at JPL. At JPL, the dominant modeling language is SysML, the Systems Modeling Language, and the dominant modeling tool is MagicDraw, a commercial tool that can produce SysML models. For this case study, I adapted our approach to architecture evolution to SysML and MagicDraw, to assess the ease of adapting our approach to off-the-shelf tools.

SysML [25] is a specialization of UML to the domain of systems engineering. It is defined as a *profile* of UML. SysML arose from collaboration, beginning in 2001, between the Object Management Group and the International Council on Systems Engineering. It was developed by a coalition of industry leaders and adopted as a standard in 2006. SysML is both a restriction and extension of UML. It is an extension in the sense that it adds new syntax and semantics beyond that of UML. It is a restriction in that it excludes many of the elements that do exist in UML, for the purpose of simplifying the language. SysML takes a subset of the diagram types from UML and repurposes them for the domain of systems engineering. The class diagrams of UML, for example, become block definition diagrams (BDDs) in SysML; composite structure diagrams become internal block diagrams (IBDs).

I used two diagram types in representing evolution states: BDDs and IBDs. In SysML, a *block* is the basic unit of system structure. A BDD shows the blocks that appear in the model; an IBD shows the internal structure of a block. I used BDDs to show the kinds of architectural components in my model and the hierarchical relationships among them; I used IBDs essentially as conventional software-architectural diagrams, to show the architectural structure (components, connectors, etc.) of a system. BDDs and IBDs are both representations of an underlying model.

I tailored my use of the diagram types to show those aspects of the architecture whose evolution we hoped to model. For each evolution state, I produced a complete model and a fixed set of diagrams, including one IBD showing the detailed internal structure of MPCS and three IBDs that served as context diagrams showing how MPCS was deployed in the three different environments (flight software development, ATLO, and operations). For this evolution, it was important to see not only the changes that would occur within MPCS, but also the changes in how MPCS interacted with other systems.

Figure 1 shows the IBD depicting the internal structure of MPCS in the initial state. This is a very data-flow-oriented representation of MPCS, which is appropriate given its nature.

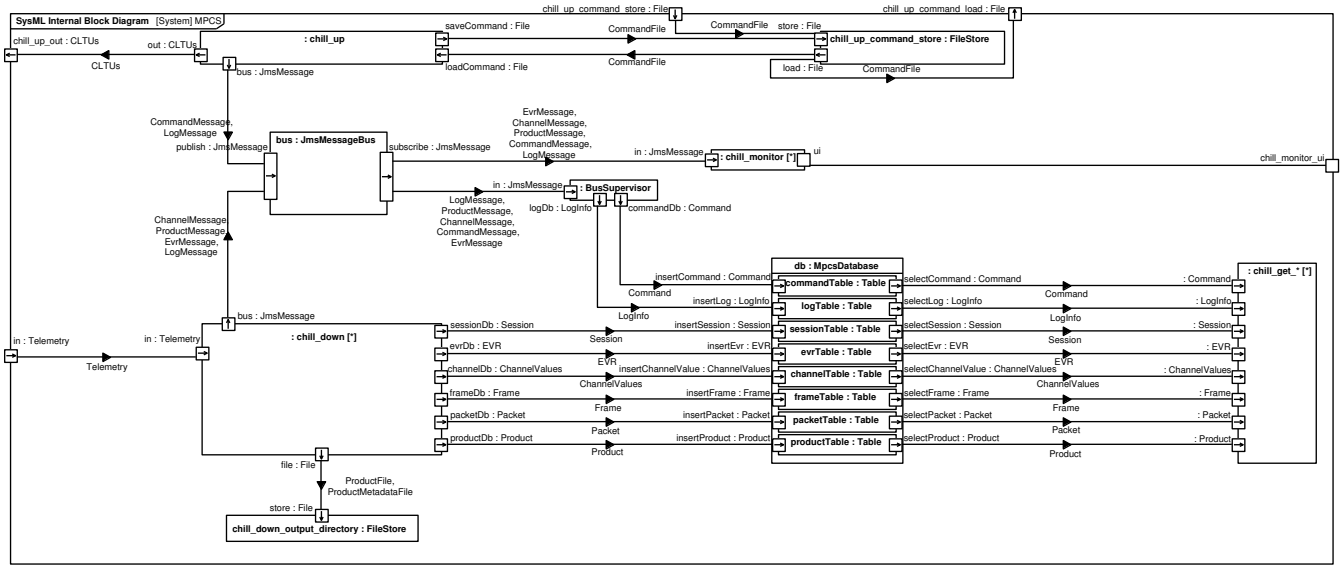


**Figure 1: IBD showing the internal structure of MPCS in the initial state**

Most previous architectural representations of MPCS at JPL have also depicted data flow connectors prominently [7–9].

In other ways, however, this representation is quite different from previous representations of MPCS. The most important is that I have drawn the boundary of MPCS more narrowly than previous representations of the system. There are several key elements that previous representations have depicted as components of MPCS but that I have represented instead as external collaborators of MPCS, namely configuration-specific components such as the flight software development simulation environment. This has two advantages. First, it makes it easy to depict MPCS without the difficulty of somehow representing all the different configurations in which MPCS can be deployed. Previous diagrams of MPCS have either addressed this issue by introducing special notation or ignored it by tacitly representing only one environment. The second advantage is that these extra components are not generally conceived of as part of MPCS anyway. Previous diagrams seem to have included them mainly for convenience and diagrammatic simplicity.

Redrawing the boundary of MPCS in this way does not eliminate the problem of representing multiple environments; it merely pushes the problem outward, so we can deal with it separately. The three different environments in each state are represented with three IBDs that serve as context diagrams, showing how MPCS interacts with external collaborators.

## 4.4 Representing the Evolution Graph in MagicDraw

Intermediate evolution states do not exist in isolation. In our model of architecture evolution, intermediate states form the vertices of an evolution graph, whose edges indicate the transitions that may occur among states.

I wished to represent the entire evolution graph in a single MagicDraw project, so that it would be possible to write constraints and analyses over the model using the constraint and analysis facilities provided by the tool. I thus placed each intermediate state in its own package. A package is a UML construct (also available in SysML) that encapsulates related entities. Placing the intermediate states in different packages isolates them while still keeping them within the same project so as to accommodate analyses of the entire evolution graph. We can represent the packages themselves in a *package diagram*; then we can represent the transitions between them by relationships among packages.

I ultimately produced an evolution graph with seven states, including the start and end states. The package diagram in figure 2 shows these states. The mainline evolution path is the simple, two-transition path from the “Initial” state to the “ICMD” state to the “Final” state. The first transition is the introduction of ICMD, and the second is the introduction of timelines. However, many alternative paths are possible.

The simplest path is to go directly from the initial state to the target state, skipping the ICMD evolution entirely—going straight to the target architecture rather than implementing integrated commanding first and timelines later. This is reasonable because the timeline evolution interacts with the ICMD evolution, and in some sense undoes part of it. The ICMD evolution rewires the commanding components of MPCS so that they communicate with CMD; the timeline evolution then rewires these same components again to communicate with the timeline management system. As is often the case with evolution paths, there are trade-offs. Going directly to the target state would be faster and cheaper than going via the ICMD waypoint. However, it would also be riskier—not only because of engineering risk due to the lack of intermediate releases, but also because of the lack of stakeholder visibility into the state of the system.

The other alternative paths in this graph emerge during the introduction of timelines. All these alternatives were developed based on discussions with project personnel and reflect different evolution options that were under consideration. Each candidate path has its own set of trade-offs.

All these possibilities, and the complex interactions between them, appear in figure 2. Each of the packages in figure 2 contains a complete representation of the architecture in that state; I have shown only the initial state (figure 1).

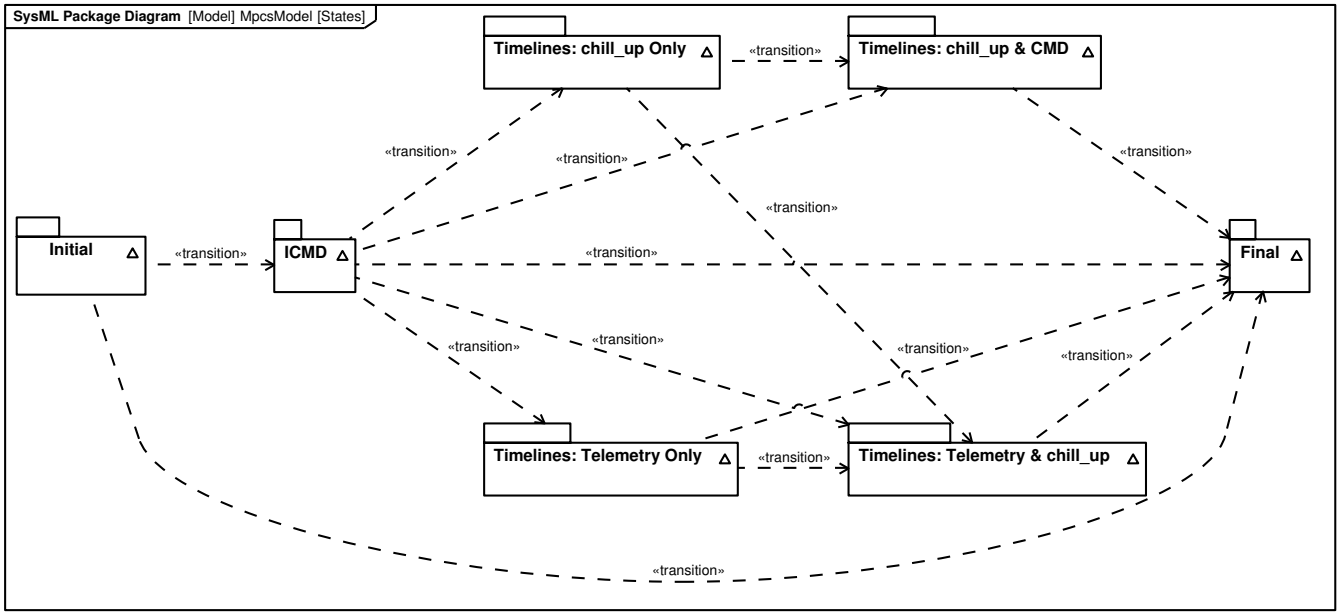


**Figure 2: Package diagram showing the evolution graph; states are represented as packages, and transitions are represented as dependencies (dashed lines)**

## 4.5 Representing Architectural Transformations

The problem with having a separate representation for each intermediate state is that it can be a maintenance nightmare. Naturally, all the states in the evolution look mostly the same, except for those pieces that are evolving. Thus, modeling the evolution graph required the production of many nearly identical packages. I could have created this evolution graph model very easily by simply cloning the initial state and modifying it. That is, after first representing the initial state, I could have copied it, pasted it, and modified it to create the next state; then done likewise for the next state; and so on. But maintaining this evolution graph would be painful; fixing an error common to many states would be very tedious.

Instead of this copy-and-paste approach, I modeled the structural transformations themselves in such a way that they could be applied automatically. Rather than generating intermediate states by hand and applying the evolution steps by hand, I specified the structural transformations needed to generate the intermediate states automatically. Then, if the initial state were to change, the intermediate states could be regenerated instantly, so fixes would need to be applied in only one place instead of many. This is analogous to the way that most revision control systems use delta encoding to store file versions (storing diffs between versions rather than a complete copy of every version of every file) or to the way that video compression works (by storing differences between frames, taking advantage of the typical similarity of nearby frames, rather than storing a complete copy of every frame).

This approach accords well with the architecture evolution framework we have developed in our research, in which architectural operations capture the structural transformations involved in evolution steps, as well as other information to support analysis. The transformations that I employed in this project fulfilled the same role here (except without providing metadata to support analysis).

I implemented these transformation specifications as macros in MagicDraw. MagicDraw supports the definition of fairly sophisticated macros that can alter both the model and the presentation of its diagrams. To do so, it exposes a rich Java API for creating and modifying model elements and presentation elements. Macros are written in a scripting language and compiled to Java bytecode.

In principle, one could use a UML transformation standard such as QVT for this purpose, rather than a proprietary API. I used macros rather than QVT for two reasons. First, MagicDraw has no built-in support for QVT (nor any other model transformation language), and although there is an official MagicDraw plug-in for QVT, it is immature and not well documented. Second, using macros allowed me to transform not only the model, but also the diagrammatic presentation of that model. With QVT I would have been limited to the former; I could have transformed the model automatically, but still would have had to update the diagrams painstakingly by hand.

The transformation specification for the entire evolution graph was 752 lines of code (excluding blank lines and comments) written in the Groovy scripting language.

## 4.6 Developing Constraints and Analyses

A useful feature of this style of implementation is that it allows for straightforward definition of constraints and analyses, which are important parts of our model of software architecture evolution. Due to stringent time limitations, I did not formally define constraints on the studied evolution nor explore the issues involved in analysis definition in depth, but the concept is straightforward.

With the entire evolution graph represented in a single model, we can represent evolution path constraints as constraints over the model, which can be expressed in OCL (the Object Constraint Language, a language for expressing constraints in UML and related languages [24]). These constraints can be automatically checked by the modeling tool.

In principle, it would be straightforward to develop a MagicDraw plug-in that allows architects to express constraints in our path constraint specification language, translates them to OCL, and checks them against the model, automatically eliminating illegal paths. However, this is left as future work.

Macros could be used similarly. Macros might be particularly useful for expressing evaluation functions, as OCL's constraint-based approach may be too rigid for analysis of evolution qualities.

Even though analyses were not a focus of this case study, it is worth considering the sorts of analyses that would be important in this domain. Many of the concerns pertaining to the alternative evolution paths appear to be based on risk. For example, as I mentioned earlier, the primary argument against evolving directly from the initial system to the target system is that it entails substantial risk. Risk is a special quality that merits special treatment in a theory of software architecture evolution. Analyzing risk is likely to entail the construction of a probability model for the evolution, which would be used to model the likelihood and potential effects of various contingencies. There is a great deal of existing work on risk modeling which we could draw on to develop a model of risk in software architecture evolution. For now, this remains as future work as well.

Other prominent concerns about this evolution include time, cost, and collaboration. These are more straightforward to model, as the time or cost of an evolution path is a simple function of the time or cost of the transitions that compose it.

These are all "business" issues rather than technical ones. But there are also technical constraints in play, and technical constraints are often simpler to analyze in purely structural terms than business constraints. In this evolution, we might have a constraint that there are always complete pathways by which commands may be uplinked to the spacecraft and telemetry downlinked; if not, there is a bug in the model.

Thus, even though detailed constraint and analysis definition was not a focus of this case study, in general terms it appears that our approach is well suited to capturing the sorts of constraints and analyses that are relevant to this case.

# 5. RELATED WORK

For a survey of research related to architecture evolution generally, see our previous paper [13]. This section presents a survey of *empirical* research in architecture evolution, to clarify the context and significance of this case study.

Previous work in this area has often relied on artificial examples—evolutions imagined in general terms but never carried out. Our previous paper [13] is in this category; we presented an example in which an ad hoc peer-to-peer architecture evolves to an architecture based on an off-the-shelf integration technology such as IBM's Message Queue Series Workflow. Although such evolutions do occur, our description was based on a general understanding of the domain rather than any specific evolution that we observed or carried out.

Other researchers have likewise relied on fictitious examples. A large body of work has been produced by Tamzalit, Le Goaer, and their colleagues [18–21, 30]. Like us, they seem to have relied on fictitious examples rather than observing real evolutions. Several of their papers [19–21] use an example of a client–server architecture based on an example given by Cheng et al. [4]. In a 2007 paper, Le Goaer & Ebraert [18] give a couple of other examples: a banking application evolving to accommodate different account types and a chat application evolving to support two kinds of users. In a more recent paper, Tamzalit & Mens [30] give yet another example, which they call EShop: an Internet shop application evolving to a client–server architecture. All these examples seem to be artificial; there is no suggestion in the papers that the evolutions described were carried out or that they were based on observation of real evolutions.

Other formal approaches to architectural reconfiguration have been lacking in empirical validation. Wermelinger & Fiadeiro [31] and Grunske [15] both present approaches for architecture reconfiguration based on graph transformations, but neither provides a substantial case study. Spitznagel & Garlan [28], in their work on connector transformation, carried out a small case study involving enhancing a Java RMI connector with Kerberos authentication. However, this was a small example in laboratory conditions, not an observation of real-world software engineering practice. In addition, Spitznagel's work focuses specifically on connector transformation and is not a general theory of architecture evolution.

Erder & Pureur [11] present a real-world case study drawn from their professional experience in the banking industry. They present the case of a loan-servicing company that was migrating from a mainframe system to an event-driven, service-oriented architecture. However, their case study is very brief and gives few specifics.

To find substantial case studies, we must look further afield than the small body of research on architecture evolution as such. One area of related work is software development planning approaches such as COCOMO. COCOMO itself was based on Boehm's study of 63 software projects, and the first chapter of his book introducing COCOMO [1] is a case study of what we would call a software architecture evolution. COCOMO (or its successor, COCOMO II) has subsequently been tested in practice numerous times, and a number of case studies describe and evaluate its application [6, 10, 22]. However, this area of work is of limited relevance here. Planning methodologies like COCOMO are focused on determining the cost of attaining some end state and do not provide ways of reasoning about sequences of development or architectural constraints. Research on the effectiveness of COCOMO and its brethren has little bearing on the usefulness of architectural approaches to evolution planning.

Outside the research literature, there are plenty of writings that describe, in practical terms, examples of what we would call architecture evolution. A number of examples can be found, for instance, in the IBM Redbooks series, which provides guidance for practitioners on topics such as migrating an Oracle database to DB2 [3], or carrying out a version-to-version WebSphere migration [32]. But such sources are aimed at characterizing a single evolution domain and do not relate the example to a general approach to architecture evolution.

It is unsurprising that there is no data on real-world application of architecture evolution tools, as the state of software engineering practice now is that few organizations use architectural approaches to evolution at all, let alone in a systematic way. Ozkaya et al. [26] interviewed software architects from a variety of organizations and found that practitioners do not use architecture-centered practices to manage evolution decisions. The planning that did exist tended to be

short-term; as one interviewee put it, "When I am planning ahead 6 months, I am being very strategic." Nonetheless, respondents did feel that architectural knowledge was important to evolution, even if underused currently.

# 6. CONCLUSION

Section 1 introduced three goals for this work. We revisit them here and discuss what conclusion may be drawn from the case study.

1. **Understand a real-world software architecture evolution problem in its natural context.** In this case study, I spent ten weeks at JPL and examined an ongoing software architecture evolution in its real-world context. As discussed in section 5, existing work in software architecture evolution has relied heavily on artificial and toy examples. While such examples do have some value in evaluating some of the characteristics of a model for architecture evolution, they provide no assurance of the its real-world applicability. By examining in detail a real-word architecture evolution—and applying previous research to this natural context—we can gain some confidence that our model of software architecture evolution, and the assumptions that we made in developing it, are compatible with software architecture as it is practiced in reality.

2. **Assess the usefulness of our framework for software architecture evolution in helping to plan evolutions and reason about trade-offs.** In interviews and discussions with JPL personnel, I found that it was indeed the case, as assumed, that architects currently lack tools to aid them in planning architecture evolution. The dominant use of architecture modeling tools at JPL is to represent existing systems, not to plan evolution. Planning for evolution is accomplished chiefly via requirements documents and informal architectural sketches of target states; there is no tool support for architectural planning. Second, many of the people I spoke to were interested in better tool support for architecture evolution. Architecture evolution was generally recognized as an important problem, and one for which little support existed. Finally, an important result of this case study was that our approach to architecture evolution was able to capture many of the real-world architectural concerns relevant to this evolution instance.

3. **Assess the ease of implementing our approach to software architecture evolution with off-the-shelf languages and tools.** Our previous work in this area has used research languages and tools almost exclusively [12]. This case study shows that benefits in evolution planning can be achieved even without special-purpose, custom tools. Our approach can be adapted to languages and tools already in place at organizations like JPL—such as SysML and MagicDraw—in a straightforward manner. This bodes well for the adoptability of our approach.

## 6.1 Generalizability

External validity is a serious methodological challenge for any case study. Generalization of case studies is quite different from generalization of studies based on statistical sampling from a population; in a case study, the goal is analytic, rather than statistical, generalization.

The results of the case study accord well with the assumption that architecture evolution is common. An important threat to validity is that JPL may differ materially from the population of software organizations to which we might like to generalize. JPL may have special qualities that make evolution particularly common there, such as the need for long-lived, multimission software. At a minimum, though, this case study shows that architecture evolution is quite common in at least some kinds of software development environments, and it gives some insight into the circumstances under which significant architecture evolution occurs (e.g., when a software system is long-lived but future requirements changes are difficult to anticipate).

The usefulness and adoptability results face similar threats to external validity. For example, there may have been something special about the modeling tools that JPL uses (SysML and MagicDraw) that make them particularly suitable for implementing our approach. This seems unlikely, however. The most popular modeling language in software engineering is UML, and the SysML modeling in this case study could have been done in UML just as easily. Similarly, other major commercial modeling tools used in software engineering have comparable features to MagicDraw.

## 6.2 Future Work

Time constraints limited the depth of the evaluation of our approach to software architecture evolution. Some aspects of our approach could not be evaluated at all; in particular, there was little time to consider constraints and analyses more than superficially. It is hoped that in future work, more complete evaluation will be possible.

More generally, further empirical work is needed to develop additional insight into the way software architecture evolution happens in the real world and to understand the kinds of tools architects need to plan evolution effectively, as well as to examine other domains in which software architecture evolution occurs.

## Acknowledgment

This research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, and was sponsored by the JPL Summer Internship Program and the National Aeronautics and Space Administration.

I would like to thank the many members of JPL's ground systems engineering section with whom I interacted. I am particularly grateful to Brian Giovannoni, Dave Santo, and Oleg Sindiy for their guidance and support on this project.

## References

[1] B. W. Boehm. *Software Engineering Economics.* Prentice Hall, Upper Saddle River, NJ, 1981.

[2] A. Cervantes. Exploring the use of a test automation framework. In *2009 IEEE Aerospace Conference Proceedings*, 2009.

[3] W.-J. Chen, K. Chen, P. Dantressangle, et al. *Oracle to DB2 Conversion Guide: Compatibility Made Easy.* Redbooks. IBM, Poughkeepsie, NY, 2009.

[4] S.-W. Cheng, D. Garlan, B. Schmerl, et al. Using architectural style as a basis for system self-repair. In

*Proceedings of the 3rd Working IEEE/IFIP Conference on Software Architecture (WICSA3)*, pages 45–59, 2002.

[5] P. Clements, F. Bachmann, L. Bass, et al. *Documenting Software Architecture: Views and Beyond*. Addison-Wesley, Upper Saddle River, NJ, second edition, 2010.

[6] L. De Rore, M. Snoeck, and G. Dedene. COCOMO II applied in a banking and insurance environment. In *Proceedings of the Software Measurement European Forum (SMEF 2006)*, pages 247–257, 2006.

[7] N. Dehghani. Mission Data Processing and Control Subsystem (MPCS). Presented at the European Ground System Architecture Workshop (ESAW'07), Darmstadt, Germany, June 13, 2007. `http://www.egos.esa.int/export/egos-web/others/Events/Workshop/ESAW-workshop-2007/day2-Session-4-0900-1040/04_Dehghani.pdf`

[8] N. Dehghani and M. Tankenson. A multi-mission event-driven component-based system for support of flight software development, ATLO, and operations first used by the Mars Science Laboratory (MSL) project. In *SpaceOps 2006 Proceedings*, 2006. `http://hdl.handle.net/2014/39852`

[9] N. Dehghani, Q. Sun, and M. Demore. NASA's 2011 Mars Science Laboratory (MSL) & supporting ground data system architecture. Presented at the European Ground System Architecture Workshop (ESAW'09), Darmstadt, Germany, May 5, 2009. `http://www.egos.esa.int/export/egos-web/others/Events/Workshop/ESAW-workshop-2009/Day1-Session-1-0920-1055/S01_03_Dehghani.pdf`

[10] R. Dillibabu and K. Krishnaiah. Cost estimation of a software product using COCOMO II.2000 model—a case study. *Int. J. Proj. Manag.*, 23(4):297–307, 2005.

[11] M. Erder and P. Pureur. Transitional architectures for enterprise evolution. *IT Pro*, 8(3):10–17, 2006.

[12] D. Garlan and B. Schmerl. Ævol: A tool for defining and planning architecture evolution. In *Proceedings of the 31st International Conference on Software Engineering (ICSE'09)*, pages 591–594. IEEE, 2009.

[13] D. Garlan, J. M. Barnes, B. Schmerl, and O. Celiku. Evolution styles: Foundations and tool support for software architecture evolution. In *Proceedings of the Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture (WICSA/ECSA'09)*, pages 131–140, 2009.

[14] W. B. Green. Multimission ground data system support of NASA's planetary program. *Acta Astronaut.*, 37:407–415, 1995.

[15] L. Grunske. Formalizing architectural refactorings as graph transformation systems. In *Proceedings of the International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD'05)*, pages 324–329, 2005.

[16] JPL. AMMOS. `http://ammos.jpl.nasa.gov/`

[17] S. C. Kurtik, J. B. Berner, and M. Levesque. Calling home in 2003: JPL roadmap to standardized TT&C customer support. In *SpaceOps 2000 Proceedings*, 2000. `http://hdl.handle.net/2014/14262`

[18] O. Le Goaer and P. Ebraert. Evolution styles: Change patterns for software evolution. In *Proceedings of the 3rd International ERCIM Symposium on Software Evolution (EVOL'07)*, pages 252–261, 2007.

[19] O. Le Goaer, M.-C. Oussalah, D. Tamzalit, and A.-D. Seriai. Evolution Shelf: Exploiting evolution styles within software architectures. In *Proceedings of the 20th International Conference on Software Engineering & Knowledge Engineering (SEKE'08)*, pages 387–392, 2008.

[20] O. Le Goaer, D. Tamzalit, M. Oussalah, and A.-D. Seriai. Evolution Shelf: Reusing evolution expertise within component-based software architectures. In *Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference (COMPSAC'08)*, pages 311–318, 2008.

[21] O. Le Goaer, D. Tamzalit, M. Oussalah, and A.-D. Seriai. Evolution styles to the rescue of architectural evolution knowledge. In *Proceedings of the International Workshop on Sharing and Reusing Architectural Knowledge (SHARK'08)*, pages 31–36, 2008.

[22] D. Milicic. Applying COCOMO II. Master's thesis, Blekinge Institute of Technology, Ronneby, Sweden, 2004.

[23] L. Needels. DSMS Information Systems Architecture (DISA) overview. Presented at JPL, Pasadena, CA, Apr. 3, 2006. `http://hdl.handle.net/2014/39174`

[24] *Object Constraint Language, Version 2.2*. OMG, Needham, MA, 2010. `http://www.omg.org/spec/OCL/2.2/`

[25] *OMG Systems Modeling Language (OMG SysML), Version 1.2*. OMG, Needham, MA, 2010. `http://www.omg.org/spec/SysML/1.2/`

[26] I. Ozkaya, P. Wallin, and J. Axelsson. Architecture knowledge management during system evolution—observations from practitioners. In *Proceedings of the ICSE Workshop on Sharing and Reusing Architectural Knowledge (SHARK'10)*, pages 52–59, 2010.

[27] A. Sanders. Innovation at JPL—GDS modernization: A case study. Presented at the Ground Systems Architecture Workshop (GSAW'10), Manhattan Beach, CA, Mar. 3, 2010. `http://csse.usc.edu/gsaw/gsaw2010/s11b/sanders.pdf`

[28] B. Spitznagel and D. Garlan. A compositional approach for constructing connectors. In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01)*, pages 148–157, 2001.

[29] K. F. Sturdevant. Cruisin' and chillin': Testing the Java-based distributed ground data system 'Chill' with CruiseControl. In *2007 IEEE Aerospace Conference Proceedings*, 2007.

[30] D. Tamzalit and T. Mens. Guiding architectural restructuring through architectural styles. In *Proceedings of the 17th IEEE International Conference and Workshops on Engineering of Computer-Based Systems (ECBS 2010)*, pages 69–78, 2010.

[31] M. Wermelinger and J. L. Fiadeiro. A graph transformation approach to software architecture reconfiguration. *Sci. Comput. Program.*, 44(2):133–155, Aug. 2002.

[32] S. S. Yu, B. Searle, D. Duffield, et al. *Migrating to WebSphere V5.0*. Redbooks. IBM, Poughkeepsie, NY, 2nd edition, 2003.