The Value of SysML Modeling During System Operations: A Case Study

Chelsea Dutenhoffer
Jet Propulsion Laboratory,
California Institute of Technology
4800 Oak Grove Dr.
Pasadena, CA 91109
818-354-4811
Chelsea.Dutenhoffer@jpl.nasa.gov

Joseph Tirona
Jet Propulsion Laboratory,
California Institute of Technology
4800 Oak Grove Dr.
Pasadena, CA 91109
818-393-8203
Joseph.F.Tirona@jpl.nasa.gov

Abstract—**System models are often touted as engineering tools that promote better understanding of systems, but these models are typically created during system design. The Ground Data System (GDS) team for the Dawn spacecraft took on a case study to see if benefits could be achieved by starting a model of a system already in operations.**

This paper focuses on the four steps the team undertook in modeling the Dawn GDS: defining a model structure, populating model elements, verifying that the model represented reality, and using the model to answer system-level questions and simplify day-to-day tasks. Throughout this paper the team outlines our thought processes and the system insights the model provided.

TABLE OF CONTENTS

1. INTRODUCTION 1
2. ESTABLISHING A MODEL.....2
3. MANAGING COMPLEXITY 6
4. COMPLEX DATA FLOWS.....8
5. FORCING CLEAR THINKING.....9
6. MODELING'S ADVANTAGES OVER TRADITIONAL APPROACHES 9
7. CONCLUSION.....10
ACKNOWLEDGEMENTS.....10
REFERENCES.....10
BIOGRAPHIES.....11

1. INTRODUCTION

The design of a GDS used to support spacecraft mission operations is never truly finished. While its basic functionality may satisfy the customers' needs initially, the system will continue to evolve. Even during the operational phase, a project's GDS will continue to be shaped as automation of data processing tools affects system performance, cyber security concerns drive adoption of new operating system patches and upgrades, and personnel changes occur both in the people using and the people maintaining the system.

The GDS for JPL's Dawn spacecraft is no exception. Planning for a simultaneous upgrade of the Dawn GDS hardware and operating system software initially proved to be a complex task, especially because those responsible for the original design were not involved in the upgrade.

In an attempt to manage the complexity of the upgrade and gain further insight into how the original system worked, the new GDS team undertook a case study to model relevant pieces of the system. The team chose to use the Systems Modeling Language (SysML) [1], a general-purpose modeling language for systems engineering applications. Goals of the model included: managing complexity, explaining complex data flows, and forcing clear and logical thinking about the system design.

While modeling methodologies like the Object-Oriented Systems Engineering Method (OOSEM) [2] often involve starting the model early in the system design phase, the Dawn case study was different in that it used an incremental, grass-roots approach that was very limited and selective in scope. This case study was undertaken more than two years into mission operations and the project had neither the resources nor a compelling need to model the complete system from beginning to end. Instead, the GDS team focused our effort on capturing pieces of the design we felt were most useful or important to specify during the course of the upgrade and using SysML to explicitly communicate this information.

Although the model began with a limited scope, its power to easily answer complex system-level questions quickly became apparent. Scripts were written which queried the model and linked previously disparate system properties with great success. As more system properties were linked, more questions were answered. This in turn prompted further questions, so more information was added. Incrementally, the model grew.

Steps were taken to verify that the model represented reality. The model was used to document changes in the system as the system evolved. The model informed design changes by answering questions like: If parts of the system go down, which machines will be affected? Which machines could be backups for each other? What is the operating cost for a given hardware configuration?

This paper will address the goals of the modeling task, make a judgment on how successfully they were achieved, and highlight some new insights into the system that resulted from the modeling process.

978-1-4673-1813-6/13/$31.00 ©2013 IEEE

Due to the sensitive nature of the data stored in the model example diagrams were created for use in this paper. They are representative examples of the diagrams used by the Dawn GDS team but were created specifically for this paper.

2. Establishing a Model

Defining a Model Structure for Software

The first step in establishing a model was to carefully design a framework for capturing information in a consistent way. Since the model's purpose was to manage complexity during GDS hardware and software upgrades, the logical place to start was by defining a "GDS software delivery" in SysML. A "GDS software delivery" already had an accepted definition before the modeling task began: an aggregation of one or more specific versions of software elements from a core set of available elements, packaged and tested to be deployed together.

A metamodel lays out the hierarchical and compositional structure of a model. It defines basic relationships between elements of given types. The team used stereotypes to define the system's hierarchy. A stereotype is defined as a method of "[extending] an existing metaclass, and … [enabling] the use of platform or domain specific terminology or notation in place of, or in addition to, the ones used for the extended metaclass". [4]

Figure 1 shows the software metamodel for this case study:

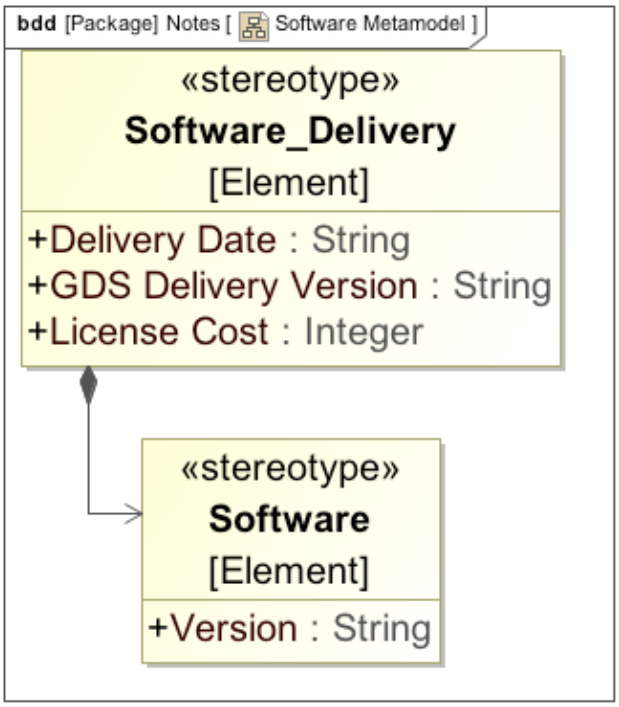

Figure 1 - Software Metamodel

Figure 1 shows that a Software_Delivery stereotype is an aggregation of Software elements. Slots for stereotype properties are also shown. For example, a Software_Delivery has a Delivery Date, a GDS Delivery Version (which represents the version number the team assigns to the whole aggregation), and a License Cost (which is related to the Operating System that the delivery is built for). The general Software element has a slot for a Version property. No specific software names have been defined in this diagram, the metamodel view simply sets up basic model elements and relationships for more specific use later.

The next step was to use the metamodel's structure to model one generic GDS delivery containing all possible software elements, as shown in Figure 2. Specific names of software elements have now been included; in this case, the names are "SoftwareA", "SoftwareB", and "SoftwareC".

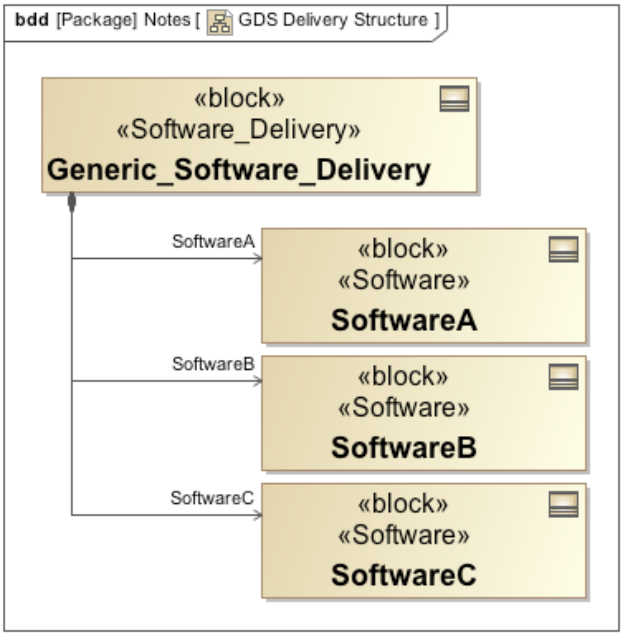

Figure 2 – Generic GDS Delivery

The elements in Figure 2 build upon the metamodel shown in Figure 1. For example, Generic_Software_Delivery is a SysML block of stereotype Software_Delivery, so it will have all the properties and relationships that the Software_Delivery stereotype has. Likewise, SoftwareA, SoftwareB, and SoftwareC are blocks of stereotype Software and will each have a Version number property. Figure 2 shows the next layer in specification after the metamodel and contains a superset of all available software.

The name on each arrow in Figure 2 represents a SysML part property of the Generic_Software_Delivery. For example, SoftwareA is a part property of Generic_Software_Delivery, meaning that some version of SoftwareA is part of the delivery. No version identifiers have yet been specified either for the complete software delivery or for any Software element's version.

With the definition of a generic GDS software delivery established, the team then added another layer to the system specification by representing a specific software delivery. The "Specific_Software_Delivery_V1" shown in Figure 3 inherits all of the generic pieces of software that make up the "Generic Software Delivery" via the generalization relationship.

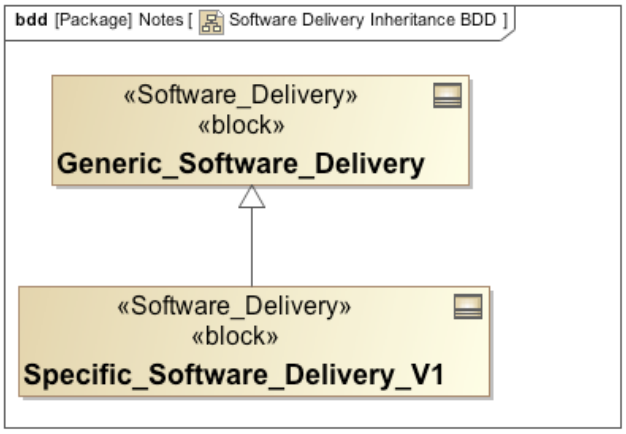

Figure 3 - Specific Software Delivery Inheritance From Generic Software Delivery

Figure 4 illustrates the use of SysML's redefinition concept. Redefinition is defined as a "change [in the] definition of an existing feature" and was used to differentiate new GDS deliveries and software versions from their previous/generic counterparts shown in Figure 2 and Figure 3. [3] For example, in Figure 4 SoftwareB_V1 redefines Software_B and is a specific part of the delivery Specific_Software_Delivery_V1. The "V1" designation added to "Specific Software Delivery" is the version number for this aggregation of software elements.

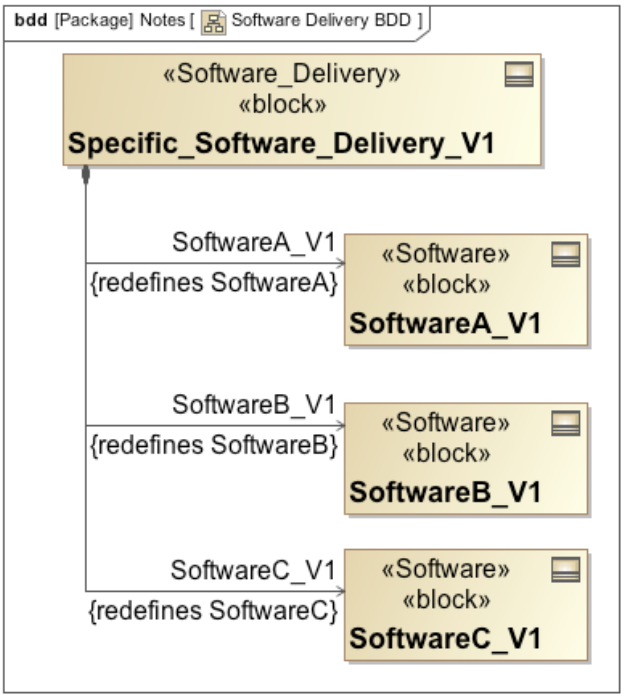

Figure 4 – Specific GDS Delivery

It then follows that each specific software delivery would inherit generic software components from "Generic Software Delivery". Although the software delivery shown in Figure 4 includes one version of each allowable software element, any given delivery is not required to contain all allowable elements. This was important because the team delivers far more partial "point" deliveries than full deliveries.

New versions of each software element are modeled as separate blocks from any other element, including prior versions of the same piece of software. This is due to the nature of the software included in GDS deliveries. Some software organizations deliver their elements as complete installs with each version, but other organizations deliver only overlays, which are installed on top of older versions of the same software. A project could use versions 1 and 3 of a software element and choose never to install version 2, so we needed a way to represent multiple versions of the same software layered on top of each other. Redefinition of properties provided the team with the freedom to more accurately capture the system.

Defining a Model Structure for Hardware

With the software modeling structure established, the next step was to define a metamodel for the GDS hardware elements. This metamodel is shown in Figure 5:

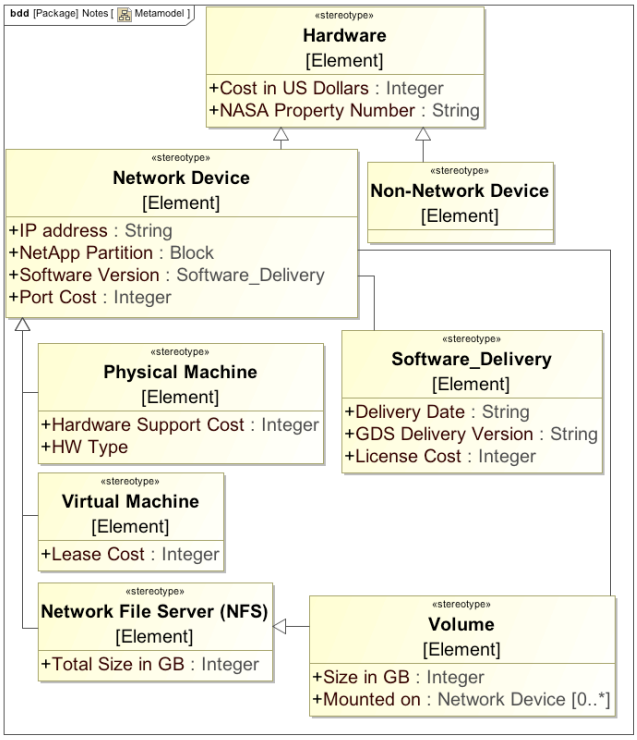

Figure 5 - Hardware Metamodel

Like the software metamodel discussed in the previous section, the hardware metamodel describes the building blocks of the system and relationships between these elements. It also allowed the team to categorize and hierarchically group sets of properties that were useful to track for different hardware element types.

For example, all NASA property is assigned a property number, so the Hardware stereotype has a slot for a value called "NASA property number" as shown in Figure 5. The Hardware stereotype is specialized into Network Device and Non-Network Device, each of these stereotypes will inherit the property number slot from its parent. Then Network Device is further specialized into Physical Machine and Virtual Machine, each of these stereotypes will inherit the property number slot as well.

Note that hardware metamodel shown in Figure 5 includes the same Software_Delivery stereotype from Figure 1, which allows mapping between hardware elements and the software installed on them. The Network Device stereotype has a slot for a Software Version; this Software Version is then set to the name of a GDS delivery. For more information see the "Linking Software and Hardware Elements" section below.

The hierarchy shown in Figure 5 was the result of careful thinking and the progression of this hierarchy shows the modelers' progression in their understanding of the GDS. At the beginning of the modeling task, the team had already been supporting the operation of this hardware for more than a year and had developed a mental model of it over time. Due to the nature of the required support, the team was used to thinking primarily about where the hardware was located. It makes sense then that initial versions of this hierarchy were location-based, separating machines that were physically at JPL from those in use at remote partner sites.

As the team came to better understand the system, we began to abstract it. At this stage the team changed the hardware metamodel hierarchy to be venue-based by separating development, test, and operational hardware regardless of its location. The team eventually found that many of the hardware properties we wanted to capture were the same across these development, test, and operations partitions. The boundaries between the hardware metamodel stereotypes were not clean enough. The team then realized that for our implementation, the best way to partition the hardware was by physical properties that did not change, like whether or not a component could access the network.

To identify the full list of value properties that should be included in the model, the team turned to JPL's existing institutional hardware databases. These databases store various parameters about each piece of hardware, and together almost completely describe each hardware component. The only information these institutional databases lack is which software versions are deployed on each piece of hardware. Initially the team planned on capturing only the most important of these properties in the model. However, since the team wanted the model to become an authoritative source for all information about the GDS system we decided that the model should include the superset of all information that already existed in these databases. This later proved to be a good decision, because the model contained diverse information that could be used to answer system-level questions the team had not originally considered.

The architecture of the hardware and software metamodels, combined with the large quantity of hardware information stored in the model itself, allowed the team to write scripts to query the model and return information. These scripts determined which pieces of hardware are connected to each other, produced lists of hardware with similar properties, and summed things like maintenance costs. Each of these will be discussed in detail in later sections of this paper.

Populating and Updating Model Elements

The first step in populating the model was to create relationships to show each of the specific software deliveries. Due to the small number of elements and simple relationships, the team populated the necessary deliveries by hand. Populating the model with hardware information proved to be a much more complicated task. Each hardware element can have dozens of properties, and the risk of humans introducing errors was too great. The team needed a different solution.

The existing institutional databases that store hardware information can output their contents as comma-separated

value (CSV) files, and the modeling tool used by the team was able to automatically populate model elements using these CSV files along with the hardware metamodel. This made initially populating the model with all the system's hardware elements fast and simple. The CSV import/export process is bi-directional, meaning CSV files can also be exported from the model and imported into institutional hardware databases in order to keep them in sync.

Information from JPL's hardware databases was imported into the model only once. From that point forward, all updates were made in the model and exported to the institutional databases as necessary. The team wanted the model to be the one authoritative source containing the superset of all information about the system, so continuing to maintain this information outside of the model would have created opportunities for information in different sources to diverge.

Each of the institutional databases provides a specific view of the system, but the source of this data is disparate and prone to inconsistencies between various databases. For example, a common inconsistency was two different databases showing conflicting IP addresses for a single computer. During the import process the team found and corrected many of these inconsistencies to make the model accurately represent the system. This points to one advantage of the model: although information in the model can be viewed in many different ways to match the existing database views, it always comes from the same source. Two views of the same model cannot have conflicting information in them. This process validated the model as an authoritative source for the project's hardware property information, which has proven to be incredibly useful for simplifying routine tasks like performing yearly hardware inventories.

In addition to the import/export features, the modeling tool the team used provided a built-in table wizard to view and edit properties in a spreadsheet format as shown in Figure 6:

#	Name	IP address	Cost in US Dollars	NASA Property Number
1	Computer9	123.123.123.120	1	NASA1
2	Computer8	123.123.123.121	23	NASA2
3	Computer7	123.123.123.122	45	NASA3
4	Computer6	123.123.123.123	54	NASA4
5	Computer5	123.123.123.124	5	NASA5
6	Computer4	123.123.123.125	54454	NASA6
7	Computer3	123.123.123.126	3	NASA7
8	Computer2	123.123.123.127	454	NASA8
9	Computer1	123.123.123.128	32	NASA9

Figure 6 - IP Address Table Wizard Example

The team was used to seeing information in a spreadsheet format similar to the table shown in Figure 6, so having this view available helped ease the transition to the model. This table is a very convenient view to show large amounts of information at one time and was used to spot-check elements to make sure they had been imported properly.

Linking Software and Hardware Elements

After the system hardware and software were both modeled, the two were connected in order to denote which GDS delivery a piece of hardware had installed on it at any given point in time, as shown in Figure 7:

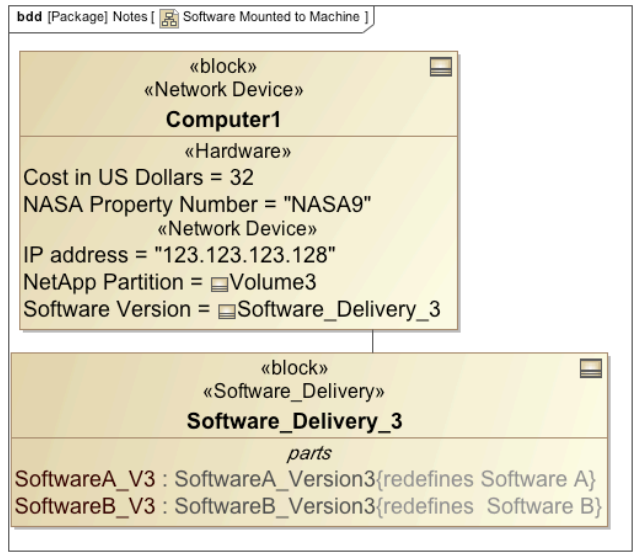

Figure 7 - Software and NFS Allocation

Figure 7 shows a hardware element of type Network Device called Computer1 that has a Software Version called Software_Delivery_3 installed on it. By looking at the Software_Delivery_3 block, a person looking at the diagram (or a model query script traversing relationships between model elements) can clearly see exactly which software elements and versions are installed on Computer1.

Note that this portion of the model is dependent upon the accuracy and depth to which the system's software and hardware were modeled. As new software deliveries became available, the modeling team needed to update these properties onto the model's hardware elements. At first it sounded like this would make updating the model after a software delivery more time-consuming, but it proved to be a good way to ensure all the hardware had been updated appropriately. Paper deployment checklists were replaced with model-generated reports. With the model elements in place, the team was now ready to verify them.

Model Verification

A model that describes a particular system is only useful if it can be proven that the model accurately describes the system. The model elements and properties need to be verified against the actual system hardware and software.

The GDS team already had an automated checkout script that would run on all the project computers and return a single XML file listing the values of several hardware and

software properties on all the machines. A similar script was written to query the model and return a second XML file in the same format as the existing checkout script output. The team ran both scripts, then looked for differences in the resulting XML files and resolved all conflicts. This process can be repeated on a regular basis to ensure that the model continues to properly reflect the system as both evolve.

Once the model was set up, populated, and verified, it was time to start using it to analyze the GDS system.

3. Managing Complexity

Viewpoints Developed to Manage Complexity

After laying the groundwork by defining the system and its elements, several viewpoints needed to be developed to provide useful information about the system. In this context, a "viewpoint" is defined as a diagram that shows a particular subset of information from the model in order to convey specific details of the system.

The important thing to note about these viewpoints is that they are products of the initial model architecture and data population. Since the subset of information being expressed has already been documented in the model, "creating a viewpoint" is often just a matter of selecting which pieces of information are important to display and dragging them onto a new diagram. As the model became more complete, more and more useful views were identified and created. This section gives examples of views that the team found to be useful, and which factors prompted their creation.

Hardware Locations

The GDS hardware and software upgrade that prompted this modeling task was the last one before the Dawn spacecraft's first science phase. The GDS team needed to ensure that an adequate, reliable, and stable hardware and software system was in place before the science phase began.

One of the first steps of the GDS hardware upgrade was to condense hardware previously located in two separate rooms down to one room. This one room had resource limitations like a limited number of power circuits and Ethernet ports. The GDS team had to determine if the room's existing resources were sufficient. At the time, the only way to do this was to trace cables under desks and behind cubicle walls to find all the power and Ethernet ports, then determine which ones were already in use. The task proved to be time-consuming and the team realized it would be useful to capture this information in the model so that it could be referred to again in the future. Figure 8 shows the resulting diagram:

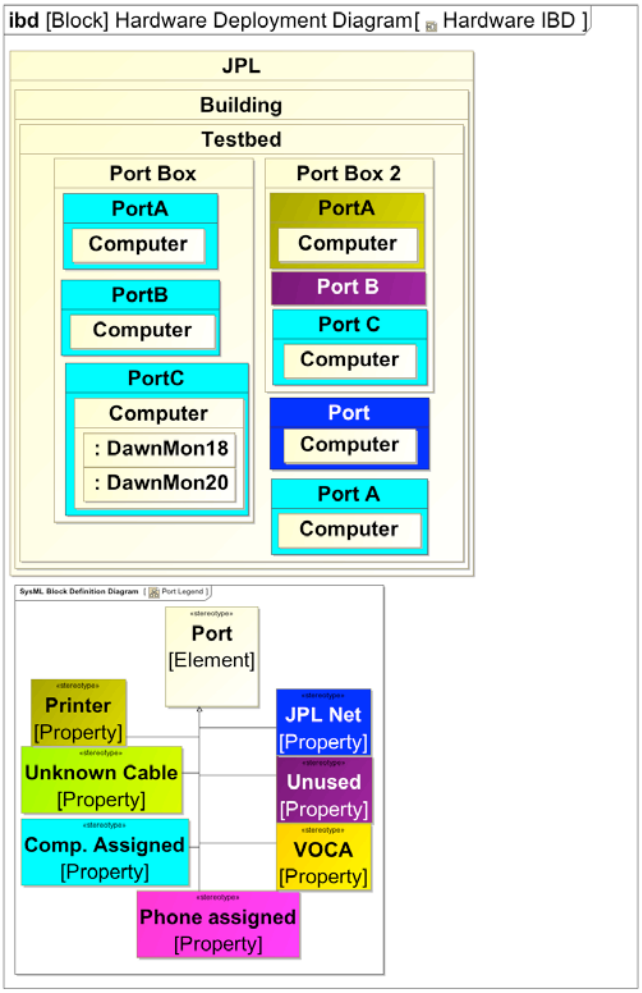

Figure 8 - Hardware Locations

The hardware location view, shown in Figure 8, came out of this hardware consolidation task and is one of the more useful views of the system. The next time users want to add hardware the GDS team can quickly determine if the room can support new hardware. Each port is automatically color-coded by the modeling tool based on its connection type. This allows the team to see at a glance what ports are in use and why.

The hardware locations internal block diagram (IBD) in Figure 8 clearly shows locations nested inside each other: JPL at the top level, and SysML blocks at the next hierarchical levels represent buildings, then rooms, then port boxes, then ports. Each port box contains three Ethernet ports. Each port represents one available Ethernet connection; hardware that uses a particular Ethernet connection is nested inside the appropriate port. Finally, for computers with monitors attached, the monitors are shown nested inside of each computer.

Traceability from a computer to its monitor is easily overlooked but is extremely useful for inventory purposes. Non-networked devices like computer monitors can be very difficult to find during the annual inventory because there is

no way to ask a monitor what its location is. Devices on the network can be logged into remotely; from there it is easy to determine an IP address and map that to an Ethernet port. Having this monitor-to-machine mapping has helped the GDS team track down several monitors quickly.

Cost Rollup

Once the team added Ethernet port tracking to the model, there was an unintentional but very useful side effect: the model could now track all the costs associated with a particular hardware configuration. Each active port has a monthly fee associated with it that varies based on the port's purpose. For example: Ethernet ports on different subnets cost different amounts, there are additional charges associated with VOCA boxes (used for radio communications), and different models of machines have different hardware and software warranty costs.

Previously these charges have been difficult to track because there was no itemized bill to the project; just one final sum that goes through the cost account was typically given. A common occurrence was that a computer was moved without the GDS team remembering to deactivate the Ethernet port and confirm that the deactivation request went through, so the project continued to get charged for an unused port without realizing it. However, if the model accurately represents the GDS system, a script can be written to sum up these expected port charges to compare them to the actual bill that the project received. This allowed the GDS team more insight into where their money was going and allowed them to catch discrepancies.

Figure 9 shows the same metamodel seen in Figure 5, but this time red boxes highlight the attributes of the hardware metamodel that are summed by the cost rollup script.

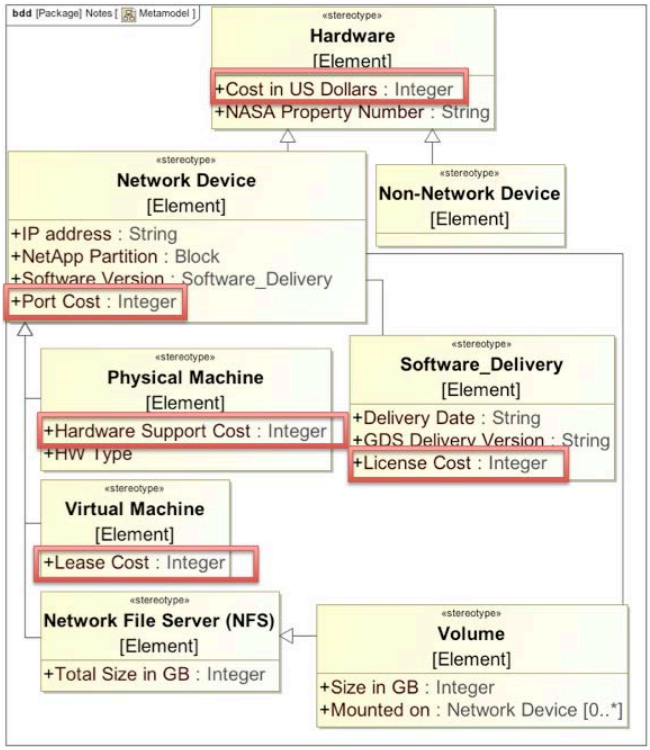

Figure 9 – Cost Script's Targets

The cost rollup script traverses the model and sums the appropriate costs depending on the machine type. For virtual machines that the project leases, the script sums the Lease Cost, Port Cost, and Software_Delivery's License Cost. For Physical Machines, the script sums Hardware Support Cost (warranty), Port Cost, and License Cost.

Scripts like the cost rollup mentioned above and the disk space query mentioned later were written in Jython. The modeling tool allowed for scripting in a preset list of languages: AppleScript, BeanShell, Groovy, JRuby, JavaScript, Jython, and QVT Operational. Jython was chosen because of the large library of Java classes that came with the tool for free and the authors' familiarity with python.

The modeling tool had intuitive ways of running these scripts as macros, which could be called easily with shortcut keystrokes. After the initial learning curve of understanding which Java libraries to call in each scenario, writing the scripts themselves took about anywhere from one to several hours, depending on the complexity of the task.

Tracking and Planning Configurations

Another big task for the GDS team is managing disk space. The project has multiple network file system (NFS) servers that house most of the disk space used by the flight team. The team knew they would need to procure additional disk space before the hardware/software upgrade, but didn't know how much would be necessary. The Dawn GDS team generally purchases a large quantity of disk space at a time, which is held in reserve and slowly allocated to the flight

team as necessary. It was important to understand up-front how much disk space might be required, so that the team could take advantage of economies of scale.

In order to prepare for the disk space purchase the GDS team went through an exercise to understand how much disk space the flight team was already using in order to predict future needs. It was clear through this exercise that there needed to be a better way of capturing disk space information, so the team decided to add the NFS servers to the model.

It was also suggested that tracking the project's required disk space throughout the life of the mission might help future projects predict their needs too. Figure 10 shows the method the team came up with for showing disk space configuration.

Figure 10 shows that each NFS server has a Total Size slot that is used to track the server's total amount of disk space. Each NFS server is divided into volumes, and each volume has an allocation of disk space. Additionally, different NFS volumes are mounted on or accessible from different computers. Capturing which computer was dependent upon which NFS server was important.

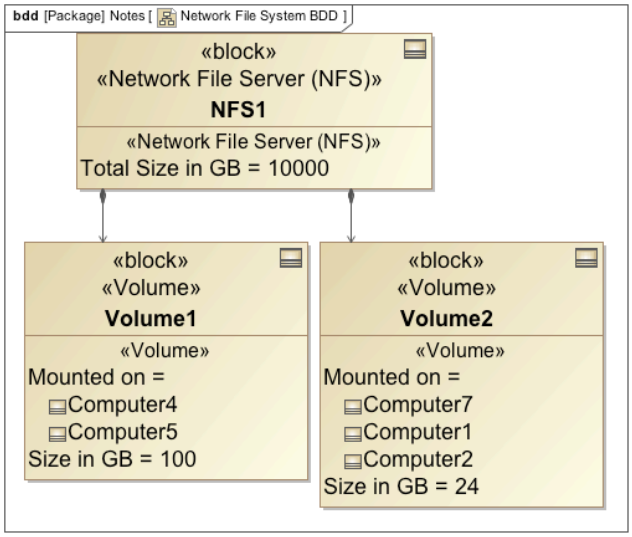

Figure 10 - NFS Partitions

Scripts were written to query the NFS model and answer three different questions: How much disk space on any given NFS server was in use, and how much total disk space did it have? If an NFS server needed to be taken offline for maintenance, which computers would be affected by the outage? Which NFS servers can any given computer access in the current GDS configuration? Each of these questions had come up at times and finding answers was always a bit challenging. The model has definitely helped with disk space management.

4. Complex Data Flows

Once the team saw that the model could successfully be queried, we began to look for other opportunities to use modeling to document the system. One of the first opportunities we found was the documentation of complex data flows.

The team knew that during the planned hardware/software upgrade some parts of the Dawn testbeds would need to be upgraded, and other parts would need to stay as they were. The team had to prove that the upgraded and legacy machines would still be able to interface with each other. The first step was to map out all the elements in the testbeds and their interfaces, as shown in Figure 11.

Figure 11 shows the connections between all the testbed elements captured on a SysML internal block diagram (IBD). Once the testbed was partially upgraded with new hardware and software, users reported performance degradation. This data flow diagram greatly assisted the GDS team in identifying where race conditions could exist or where software configurations needed to be adjusted to account for the new equipment.

Capturing data flows in the model has many advantages over a basic drawing tool. For example, the model can perform type-checks between interfaces, so if a modeler attempts to draw a connection between two incompatible ports, for example 1553 and Ethernet, the model will flag it.

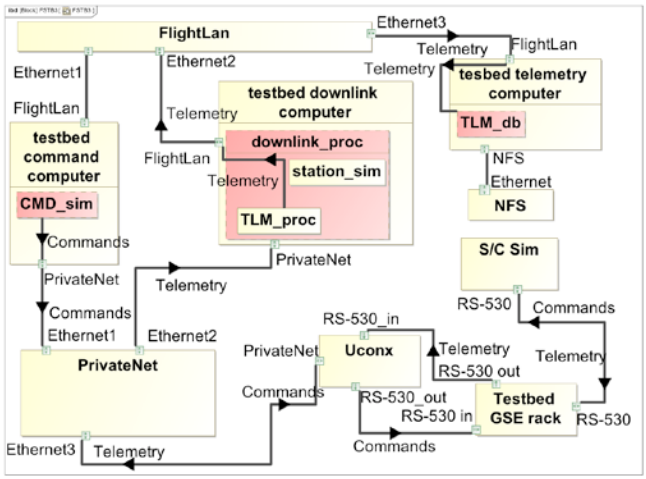

Figure 11 – Testbed Data Flows

Another advantage of modeling over other drawing methods is that the model already contains information about all the hardware and software the project uses, and that information draws its truth from a single source. As computers' properties like hostnames or IP addresses change, those changes will be automatically reflected in all diagrams. This both reduces the amount of work necessary to keep the system description diagrams up-to-date and also reduces the likelihood of inaccuracies since there are more opportunities to notice a given mistake. Dawn's prime mission is eight years long, so there is risk of documentation becoming out-

of-date. Capturing data flows in the model mitigates risk.

5. Forcing Clear Thinking

Eventually it felt natural for the team to turn to the model whenever we ran into anything needing further clarification. The act of modeling complex systems often forces the modeler to ask questions that he or she never knew needed asking. An example of this is how various pieces of software on the same computer interact with each other.

Due to way the Dawn GDS has evolved over time, there are a few cases where one tool is built specifically to call a second tool with certain options/arguments. From time to time, users will report a bug with the tool they're using, but upon further investigation, the GDS team will realize the bug actually lies in a completely separate tool. Tracing through all these tools can be complicated.

It is even more challenging when users request a new feature be added to one tool, but the GDS team finds that the change really has to be made in a different program. Now the developer needs to understand how all the tools relate to one another in case the new change breaks something else. This process requires the GDS team to think clearly about the interfaces between a set of programs, which is something the modeling process is good at promoting. An example diagram is shown in Figure 12.

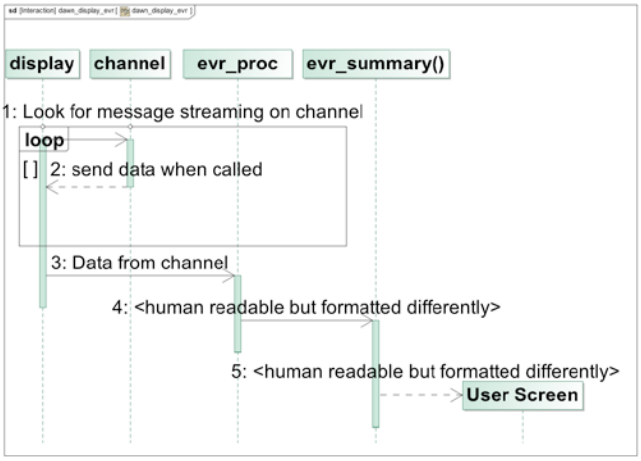

Figure 12 – Software Sequence Diagram

The SysML sequence diagram in Figure 12 shows how one software program calls a second, which then calls a third. The descriptions on the arrows show what kind of information is being passed between each program. Using SysML forces the modeler to use unambiguous syntax.

6. Advantages of Modeling Over Traditional Approaches

When describing any system as complex as a spacecraft GDS it will always be necessary to create multiple views for different audiences and purposes. These views are typically created by several different authors using presentation or drawing tools, so it is common for each to use their own style conventions. A single arrow symbol can represent hierarchy, data flow, control flow, etc. depending on which diagram you look at or who created it. It's up to the reader to determine what each symbol means on each diagram and to make sure their interpretation is correct. Instead, the use of a standardized modeling language like UML or SysML imposes a standard convention so that no matter which diagram the reader is looking at, the meaning of each symbol is unambiguous.

When using presentation or drawing tools to create system views, one big challenge is maintaining consistency across all the views, particularly since the system inevitably changes over time. The advantage of a modeling tool over these presentation or drawing tools is the model itself.

When diagrams are created using presentation or drawing tools there is no connection between each view, which allows for inconsistency. However, when using a modeling tool, all the diagrams utilize the same underlying model elements and relationships no matter who creates the diagram. If element names or relationships change over time, those changes are automatically propagated to all system views. This reduces both the amount of work necessary to keep the views up-to-date and the likelihood of inaccuracy. Just as there is only one GDS system for a given mission, there is also only one self-consistent model representation of that system.

Traditional presentation or drawing tools also have no understanding of their system elements. Through the use of metamodels, a modeling tool understands basic information about the system being described and can be the first line of defense against inconsistency. For example the modeling tool can perform type-checks between interfaces, so if a modeler attempts to draw a connection between two incompatible ports, like 1553 and Ethernet, the tool will flag it.

During operations many GDS system engineers rely on spreadsheets listing hardware properties to track their systems. Spreadsheets are useful for viewing large quantities of data at once or for passing information between teams, however they also make it easy to miss mistakes or inconsistencies. When different teams own different spreadsheets that each describe part of a single system, things get out of sync very quickly. A model enforces consistency, but users are not forced to choose between the traditional spreadsheet and a model. Once information is in the model, generating an editable spreadsheet view from that single source of truth is trivial, as was previously shown in Figure 6.

Dawn's prime mission is eight years long, so there is risk of the GDS system documentation becoming out-of-date. The transition of personnel over the course of the mission makes it even more important to maintain a single accurate representation of the system. Capturing this information in the model mitigates this risk.

7. CONCLUSION

Overall the Dawn GDS modeling task was a success. Initially setting up and revising the hardware and software metamodels took the team a fair amount of thought, but once the metamodels were in place model elements were quickly populated from existing databases. The process of importing model elements uncovered many inconsistencies in the existing hardware databases, which were then resolved.

Once the model was populated, it was used with great success to help the GDS team manage complexity. A diagram was established to track the locations of various pieces of hardware as well as building resources. Scripts were written to query information out of the model and give the team more insight into costs and dependencies of the project's computers on other servers.

As the team became more comfortable with the model, we continued to find new ways to make our job easier. The model was used to understand and document complex data flows within the system, and also to force the GDS team to think clearly about the system's interfaces.

The team's approach of selective modeling as time and money allowed worked well. Modeling the GDS piece by piece provided incremental growth in functionality and the team ended up with a fairly complete model by the end of the study. Based on the insight the team gained through this modeling process, we believe system modeling earlier in the mission definitely would have been beneficial.

The team believes this case study shows that with reasonable goals in mind, even a small team without a significant amount of money to dedicate can see benefits of system modeling. Most of the modeling and scripting in this study was performed by an academic part time employee working 10-20 hours per week over the course of one year.

The team continues to use and refine the Dawn GDS model. More work should be done to explore how to automatically audit the model's contents against the actual hardware in use on a regular basis. At the moment verifying that the model matches reality is one of the most onerous aspects of maintaining the model.

Another goal of the team's future modeling work is to generate views that will be used to educate new members coming onto the GDS team. Most system description views generated during the mission's design have become outdated and would not be of much benefit for people new to the project.

Lastly, parts of this GDS model are relevant to other missions using similar software deliveries and hardware. Being able to modularize the model and export it as a baseline for other projects would be a key goal to strive for in reducing the amount of redundant modeling work.

ACKNOWLEDGEMENTS

The research described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

REFERENCES

[1] OMG SysML Specification v1.3. [June 2012] http://www.sysml.org/specs/

[2] Lykins, Howard; Friedenthal, Sanford and Abraham Meilich, "Adapting UML for an Object-Oriented Systems Engineering Method (OOSEM)," *Proceedings of the INCOSE 2000 International Symposium*, Minneapolis, MN, Jul. 2000.

[3] Friedenthal, Sanford; Moore, Alan; and Rick Steiner. *A Practical Guide to SysML*. Burlington, MA: ELSEVIEW Inc., 2009. Print.

[4] OMG UML Specification v.2.4.1. [August 2011] http://www.omg.org/spec/UML/2.4.1

BIOGRAPHIES

Chelsea Dutenhoffer has been the Ground Data Systems Engineer on the Dawn project for the past 4 years. Her research interests include the application of model-based systems engineering techniques and methods for encouraging innovation in system design. She received a B.S. in Aerospace Engineering from Embry-Riddle Aeronautical University and is currently pursuing an M.S. in Mechanical Engineering with a concentration in Engineering Design at the University of Southern California.

Joseph Tirona is a member of the System Verification and Validation Group at JPL. He has used MBSE to model various aspects of Ground Data Systems on a number of flight projects, including the Mars Science Laboratory (MSL), Dawn, and the Orion Multi-Purpose Crew Vehicle Program. He is currently working as a Systems Engineer doing V&V on the Soil Moisture Active-Passive (SMAP) mission. He received a B.S. in Mechanical Engineering from California Polytechnic University, Pomona, where he is a graduate student in the Mechanical Engineering Department.