A Comparison Of Model-Based Methodology in Robotic Development (August 2016)

Paul M. Calhoun, *Member, IEEE*

***Abstract*— This paper investigates the relative usefulness of UML and SysML based process frameworks in robotics. Specific emphasis is placed on interfaces, which are handled very differently between the two when dealing with mechatronics and software which directly interacts with its environment via hardware. While UML and SysML are modeling languages and so not themselves processes, they are the backbone of the processes that use them, and so by taking examples of projects using frameworks based on their respective modeling languages we can see the relative merits of each in the robotics setting. This will be done by examining two robotics projects which used modeling methods derived from UML and SysML and comparing their end result with specific emphasis on how they handled certain factors which separate robotics from other applications. The UML project is a service robot and the SysML project is a satellite servicing spacecraft. An additional case study is presented for SysML, as the project examined did not present all the artifacts they used.**

I. Introduction

MODELING has become a major method of software development, with several very popular frameworks and two large product lines being based on the concept of models in UML and languages derived from it. Every phase of development has seen a model-based diagram created, in an attempt to turn products and projects into something that can be described using a series of diagrams which show the relationship between all parts of the process and product. The end goal being that a project can start with the highest level vision and use case diagrams and decompose those down into deeper levels of architecture, data flow, sequence, and state until the entire project has been mapped and the product has been described to the level where actual development is almost an afterthought. Most development does not rely quite so heavily on models as to try to diagram every single action they take, and the field of robotics has so far been less inclined towards modeling than other software-intensive applications [1].

Robotics as a field has lagged behind in this type of development framework because models are slow to develop or difficult to tailor to applications where the software influences its environment through hardware. Robotics development is still closer to early 1990s software development practices in its emphasis on results over process, and on quickly putting together a solution that fits the single problem a customer has. As such, most projects use either Agile or code-driven type processes, and those that do put into place larger scale process frameworks usually try to expand Agile methods into the scrum-of-scrum method rather than using model-based or document-driven frameworks. Some of this may be to do with the large amount of influence the Defense industry has on robotics. In the author's personal experience, the non-Agile framework that seems to be most recognizable is the Department of Defense Architecture Framework (DoDAF) and the Defense Acquisition Framework model. The former is based on UML and the latter on several different process frameworks including Spiral and Iterative types, but what they have in common is the characteristically conservative approach to development which requires large quantities of documentation and usually very long lead times [2].

The first problem is the more important one to address, as no matter how easy or fast a framework is, it needs to be useful for the application. The criterion therefore is completeness – that is whether the tool or framework can encompass the specific needs of a perceive-act-perceive style of system which influences its environment and so is able to provide its own inputs via direct interaction. The rest of this paper will describe two projects that used UML and SysML based frameworks, along with case studies which expand on their use. It will investigate the differences between how the two projects implemented their model and the relative merits of the implementation. Finally it will suggest further improvements and which of the two systems seemed to contain more useful material and so needed less data to be expressed outside the model.

II. Modeling Languages and Process Frameworks

The two modeling languages to be used are UML (Unified Modeling Language) and SysML (Systems Modeling Language). The two are quite similar, being structured sets of diagrams which are used to express different aspects of a system and the project that produces it. Each class, conceptual package, and even the distribution of data to its end target is shown, alongside use cases, activities, state transitions, data flows, and interfaces. While most types of diagrams can stand alone, they are intended to be part of a conceptual hierarchy

Submitted August 5, 2016 as the final paper in Managing Software Development course, SCS, CMU

Paul Calhoun is with the Robotics Institute, SCS, CMU, The Robotics Institute 5000 Forbes Avenue Pittsburgh PA 15213-3890 (e-mail: pcalhoun@andrew.cmu.edu).

which decomposes the requirements as they are developed and expressed in use cases down through the objects, states, data, and finally the sequence of events that the system goes through in each action it takes. At first it may seem as if some of these concepts would break down in an autonomous robotic platform, but the robot usually has a goal and so while the state transitions and sequences may be more repetitive or at least harder to predict at the start, they still have a defined start and end position and can be conceptually encompassed.

It is not within the scope of this document to go into a detailed analysis of UML or SysML, though more will be discussed in the COMET (***C***oncurrent ***O***bject ***M***odeling and Architectural Design M***et***hod) UML framework description, OOSEM (Object-Oriented System Engineering Method) SysML framework description, and in the project use cases.

A. UML

UML is the basis for most modeling languages now in use. It contains 15 separate modeling diagrams(Fig. 1), and is often tailored by the specific framework that uses it. At the core is the concept that the system can be shown as more and more granulated levels of data flow, from user input and system output down to individual objects calling functions from each other and states changing internally. Interaction with the outside world is shown in terms of data being *deployed* or sent to outside systems and users, and interaction is strictly considered in terms of data. [3] A common criticism of UML even at the highest level is that it lacks quantifiable results for the system, in that there is no flow, shape, or diagram that shows the performance parameters of the system, only that data flows from A to B via a process.

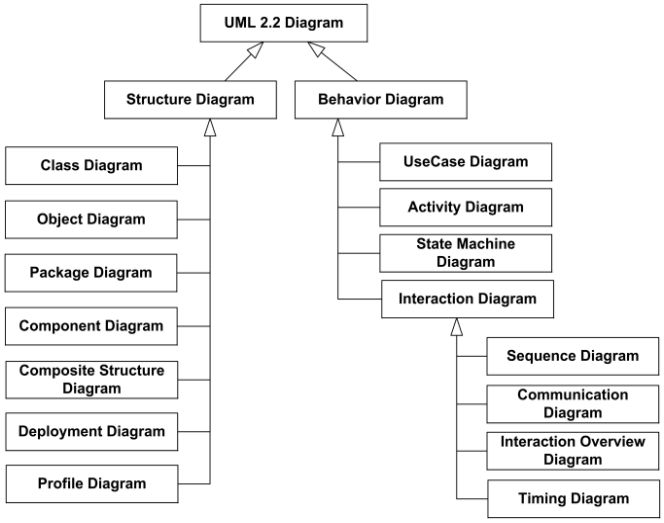

Figure 1 - UML 2.2 Diagrams Overview [3]

B. SysML

SysML is an extension – and to some extent simplification – of UML (Fig. 2). It loses some of the detailed granularity in order to be able to treat software and hardware the same way within the system. It has 9 modeling diagrams, two of which are new and 3 are modified versions of their UML counterparts [1]. The rest are unchanged from UML. SysML's largest change is that it treats hardware components as blocks alongside software components, and it has a parametric diagram which allows for plugging in equations and numeric representations of how the system will function, allowing for performance based requirements, goals, and tests to be included as part of the model. Instead of sending data to nodes in the deployment model, SysML uses the deployment model to show data flowing between software and hardware blocks. Requirements are recorded as textual items which are then related to use cases and parametric artifacts to show how performance and usage is related to each requirement [4].

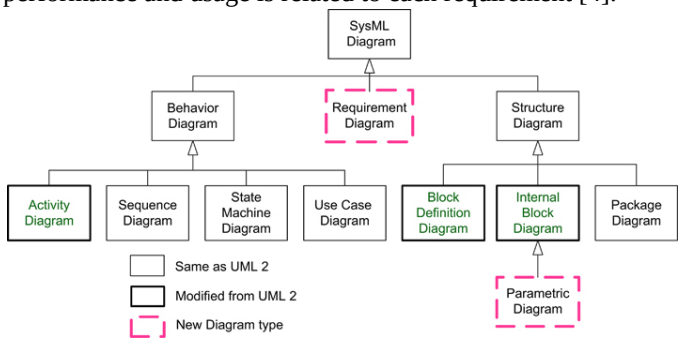

Figure 2 - Diagram relationships between UML and SysML [1]

C. COMET

COMET is a UML-based framework developed by Hassan Gomaa at George Mason University which seeks to model real-time applications which are state-dependent in their inputs and outputs. It is a highly simplified method with two phases which focus heavily on being able to model a system that reacts to input from users and its environment [5].

The requirements phase starts in the usual way of producing Use Case diagrams with actors that represent either users or input-output devices like sensors or timers. Then it models the static system with the system itself as an aggregate of classes and objects, and the external environment as a set of classes depending on how it interacts with the system. The dynamic model mixes the use cases with sequence diagrams and state charts, showing how each use case can proceed depending on the stimuli which change which state the system is in. It does so by adding a state dependent control object to the use case, which breaks up each case into a set of cases which are the branching possibilities of the system in each combination of valid states. The interaction between actors is shown on a collaboration diagram. For example the interaction of a user with a sensor to alter the state of the system.

The design phase starts by developing a consolidated collaboration diagram which shows all interactions across all use cases. This is meant to provide a complete description of all message communication. There is then a task structuring subphase which maps the objects to tasks they perform, and active interfaces also become tasks. After this, the composite tasks are split up, synchronization issues are addressed, connector classes are designed, and internal task sequencing is defined. COMET uses real-time scheduling theory and event

sequence analysis to analyze performance of the design [5].

More detail on COMET will be provided in the case study of its application to robotics, as the author of the COMET method does not elucidate any further than this in his papers on the subject.

D. OOSEM

OOSEM is a SysML based framework developed by Lockheed-Martin, which seeks to model complex hardware-software systems while maintaining flexibility with respect to evolving requirements and mission situations. It has 5 main phases and 5 common sub-activities which make up the process of engineering a system [6] (Fig. 3).

OOSEM uses SysML to bring an object-oriented approach to hardware/software development in a large scale application. Concepts OOSEM adds to typical object-oriented system engineering include causal analysis and requirements variation analysis as well as an enterprise model [7]. To support the enterprise level conceptualization of the process, it defines a sub-activity 'Reuse Opportunity Analysis' which looks at how the system being developed can be used elsewhere in the company, and to support the process being used across the entire life cycle there are also activities for 'Capture Domain & Assumptions' and 'Optimize & Evaluate Alternatives' showing that the framework can begin before the contract is even awarded [6].

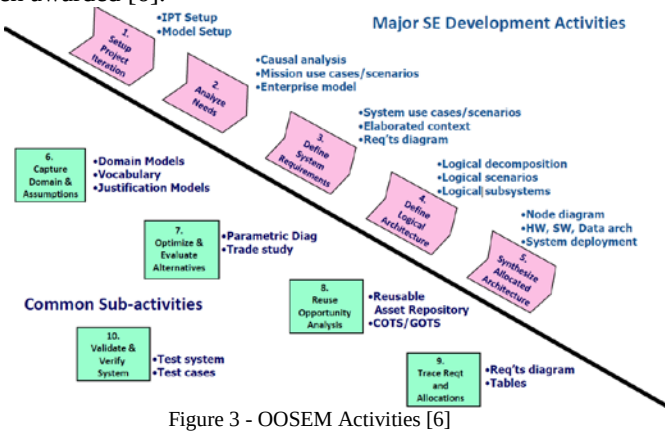

Figure 3 - OOSEM Activities [6]

The main phases are setup project iteration where the IPT (integrated product team) and model are setup, forming the team and defining the structure of the model [6]. There is a needs analysis phase where the current enterprise of the customer is captured, then the enterprise as it will operate with the new product integrated into its business. Mission requirements are captured and business use cases are developed. The third phase starts on system use cases and requirements, with a requirements variation analysis where it is determined how likely it is for each requirement to change during the project (ex. an interface being changed due to evolving needs), and what risk that poses along with mitigations including designing the system to accommodate the change.

From here on, the design decisions are made. In the fourth phase, the logical architecture is defined which includes components, data flow between them, and conceptual subsystems. In the fifth and final phase, the allocated architecture is synthesized, modeling the relationship between real components like sensors, software, and data, defining resource distribution, and allocating logical components to hardware, software, etc [7].

During this, the five sub-activities are carried out in each phase. Capturing domain & assumptions, optimizing & evaluating alternatives, reuse opportunity analysis, tracing requirements, and verification & validation of the system. The concurrent nature of these activities means that each step in development is carried out with the understanding of how it impacts the customer's needs and future projects' usage of the system.

III. CASES

The two cases that will form the backbone of this analysis are a COMET UML project to build a service robot for elder care and an OOSEM SysML project building an orbital robot that services satellites. Supporting cases will be brought in at the end to show elements that might have been left out due to tailoring of documentation by the process or application.

A. T-Rot

The project to make this robot started out as an outgrowth of the Public Service Robot (PSR) project at the Korea Institute of Science and Technology (KIST). The authors of the paper this case is taken from sought to expand the capabilities of their PSR model to be useful in the often unpredictable environment of providing services to the elderly. Their primary system goals for the fully functional product were a robot that could understand and react to voice commands, know its name when it was spoken, localize where the speaker is, pick up objects, and autonomously navigate its environment without collision and in a reasonable time frame. The goal of the authors' part of the project was autonomous navigation without collision [8].

They had 150 developers in 10 locations, each location working on a different part of the problem. With that level of distribution, integration was going to be very difficult, as would almost every other activity that required one function to interact with another, including requirements tracking. They decided to use an object oriented model based approach, and settled on COMET as their framework because of its emphasis on real-time dynamic process models. In the course of their paper on the topic, they focused on the development of the function they were responsible for – autonomous navigation.

The use case (Fig. 4) for their part of the system was quite simple; two functions and two actors. The navigation function was acted on by the commander and the obstacle avoidance function extended the navigation function and was acted on by a clock. Their logical process was that the navigation system would be given commands from a commander node – either the user or a decision making process which determines what the robot needs to do based on its environment – and the collision avoidance process checks the sensors four times more often than the navigation system does (hence the need for it to be acted on by a clock) and sends an emergency stop to the system if an obstacle that was not planned for is in the

way. If the obstacle is no longer sensed, the avoidance function turns off the emergency stop. The use case makes no provision for an obstacle that remains and for re-planning.

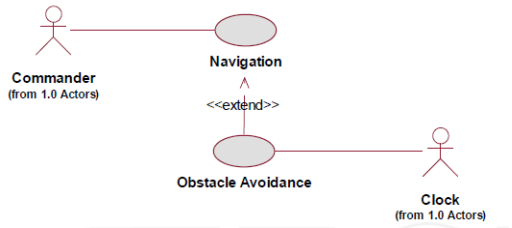

Figure 4 – T-Rot Navigation Use Case [8]

In the static modeling phase, they had to model the interfaces between the external environment and the robot, and the structure of the system. To do this, they used system, external input, external output, external timer, and external user defined blocks. For instance, the navigation system block took input from the sensor input device block and sent and output to the wheel actuator external output device block. This treats interactive hardware as data input and output, with the robot a nominal/ideal operation of sensors and motors – and the model treating them entirely as data input and output devices which feed back into the system to update maps and position information.

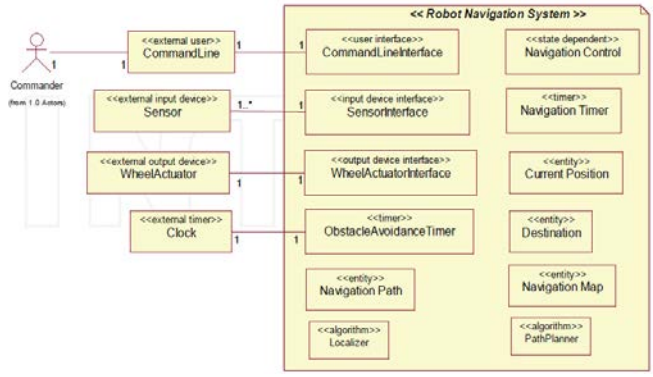

Figure 5 - T-Rot Structuring Class Diagram for Navigation System [8]

Something noteworthy about the modeling method in Figure 5 has blocks, but that the boundary between the navigation system's internal blocks and the external blocks which are linked to them is made using UML's comment shape. This is because this is a diagram specified in COMET that does not exist in UML, though I am unsure why they did not use the deployment diagram as it shows interfaces between the system and external devices.

In the dynamic modeling phase, they developed the collaboration diagram for navigation (Fig. 6) and obstacle avoidance. It is interesting to note that they declined to make a different chart for each state / message, so this chart as a number x.yy where x is the message which corresponds to a state. Due to the nonlinearity introduced by this, it can be difficult to read this chart, and it is also worth noting that the navigation diagram does not include the obstacle avoidance function at all, despite obstacle avoidance extending the navigation function. This is because COMET splits them up in this diagram as conceptually different subsystems and we see them together in the consolidated collaboration diagram (Fig. 8), which is part of the phase following dynamic modeling. The collaboration diagrams are split into the conceptual layers for ease of reading and understanding of what each set of blocks supports.

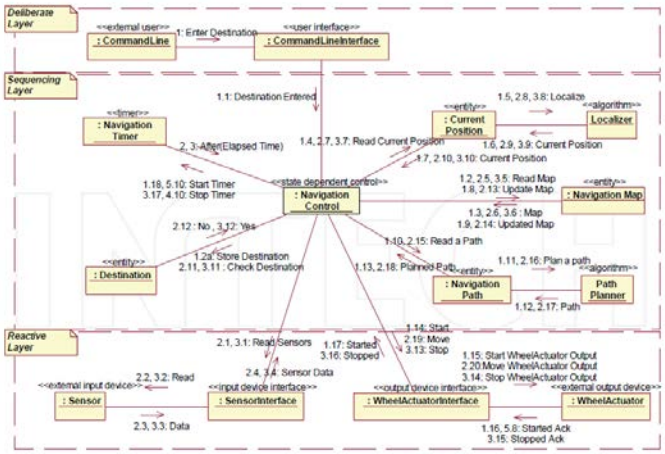

Figure 6 - Collaboration Diagram of Navigation System [8]

In addition to the collaboration diagram, there is a state chart (Fig. 7) for each function, which is referenced in the consolidated collaboration diagram. The state changes are labeled with what message in the collaboration diagram results in the state change. The diagrams are made more readable by the understanding that in the COMET framework, state changes only occur if a message is sent to the state control block [5] (in this case, Navigation Control), and so all messages on the state chart are messages that can be found entering or exiting the Navigation Control block.

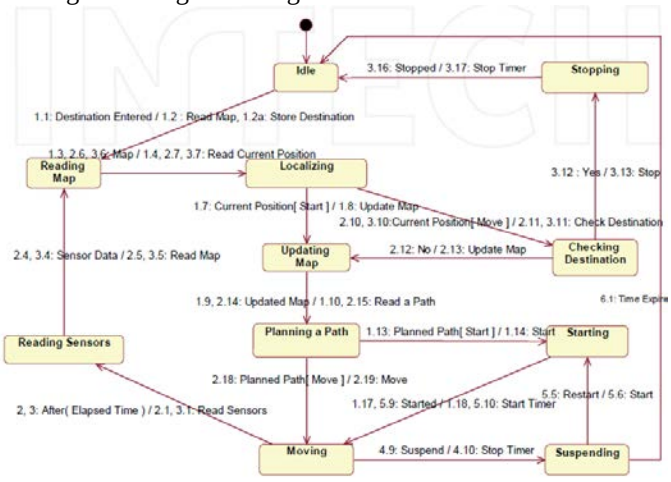

Figure 7 - State Chart of Navigation System [8]

The consolidated collaboration diagram (Fig. 8) shows the two functions of Navigation and Obstacle Avoidance together. What's interesting in this case is that the Obstacle Avoidance never communicates directly with Navigation, so it cannot execute a state change except by altering the messages coming from nodes that do. More interesting is that it can stop the wheels but that only indirectly changes the state to 'stopped' after a second message from the wheel actuator interface tells the Navigation Control node that the wheels are no longer moving.

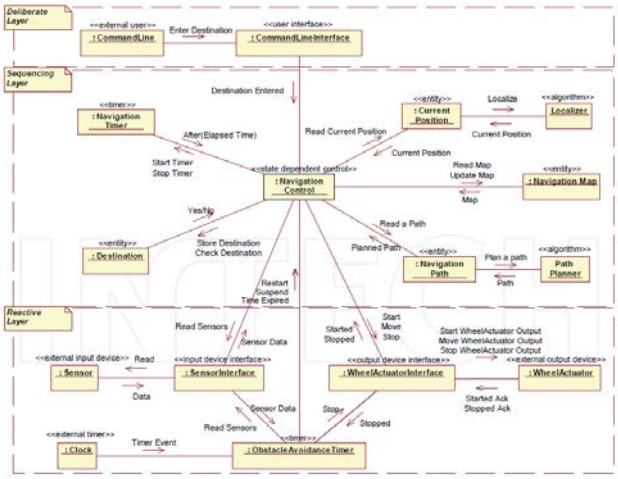

Figure 8 - Consolidated Collaboration Diagram [8]

The tasks were then structured – a set of diagrams which effectively define the messages in the collaboration diagram with their data type and flow direction – and a detailed software design, which is where the COMET concept of a message coordinator is added (Fig. 9). The message coordinator is now the gatekeeper, which processes what messages pass where and packaging them into Events, which are sent to the Navigation Control, which in the detailed software design is now only a recorder of states and not a decision maker or a place where messages pass through. The environment itself is a data abstraction in COMET UML. There is also a task event diagram, which is effectively the sequence diagram for the tasks down in Figure 9. This diagram is not part of COMET, which prescribes pseudocode rather than a diagram, but the T-Rot system engineers found a sequence diagram to be more understandable.

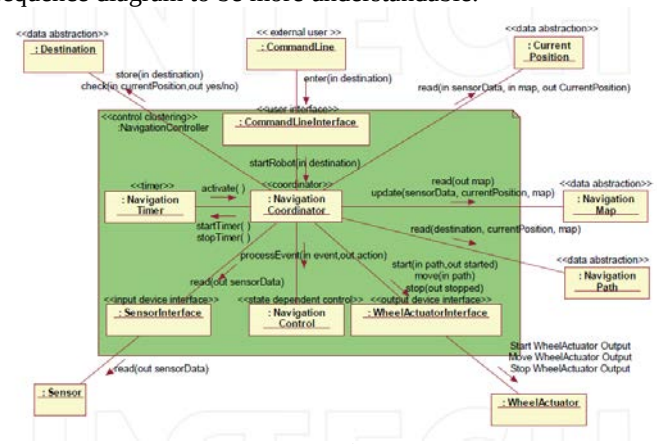

Figure 9 - Detailed Software Design [8]

The task sequence is the final stage of COMET, after which the system is implemented in code. The goal of this project was not to have a framework for the life cycle, but instead a means of easing communications between geographically and functionally separate teams. In this, it is not quite a framework but a way to make informal hacker-type development processes work with such a large group. It served as a requirements repository, design database, and configuration management method for the system with the plan of integrating through data flow rather than trying to plan out specific interfaces. For this purpose, COMET worked well for the team and they conclude by saying that their project is on course and that the models are helping to keep all the function groups working together towards a single unified system, though as of 2010 T-Rot had been retired in favor of CIROS, and no further papers have been published.

B. INVERITAS

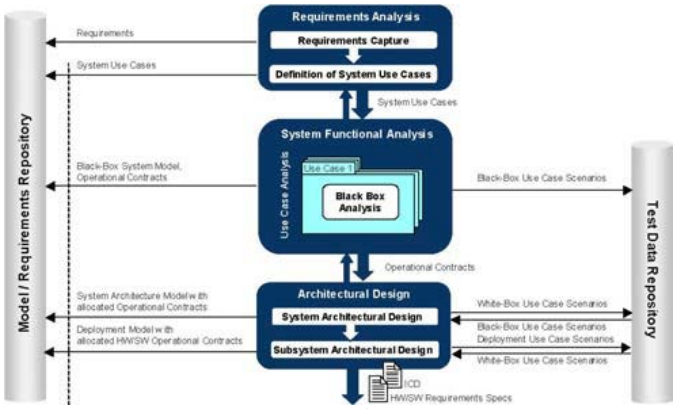

Figure 10 - Activity Flow for INVERITAS [9]

The Innovative Technologies for Relative Navigation and Capture of Autonomous Systems (INVERITAS) is a prototype rendezvous and capture system to service or retire orbital resources. It has a multi-modal sensor system (MMS) and controller interface, as well as tightly coupled sensor, maneuver, and manipulator subsystems. It inspects, maintains, repairs, and de-orbits satellites, collectively known as On-Orbit Servicing (OOS). In their paper on the system engineering process used in the INVERITAS project, the authors focus on the MMS aspect. Their chosen language was SysML, and their framework was OOSEM guided heavily by the automated modeling tools within Rational Rhapsody [9], ignoring most of enterprise level aspects of OOSEM (Fig. 10).

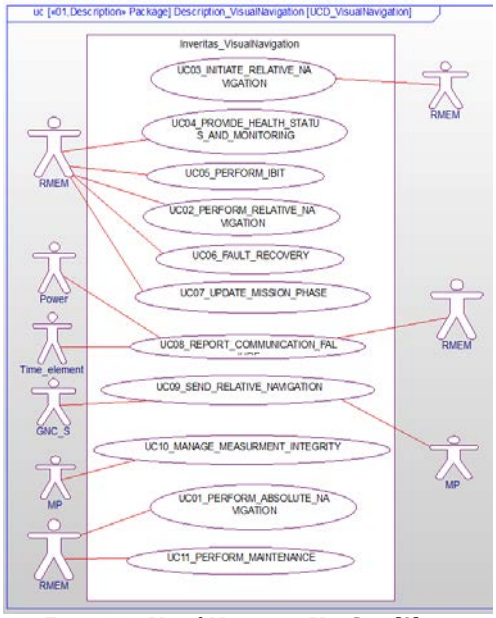

Figure 11 – Visual Navigation Use Case [9]

They used the standard decomposition from SysML and use cases (Fig. 11) which treat actors as being elements entirely outside of the system. Each subsystem is also packaged with its requirements, the requirements aggregating up into the highest level where they all appear. [9] This helps to maintain traceability as no requirement can exist in the full system without being tied to a subsystem/function, and in the same manner all functions carry their own use cases and scenarios along with their current configurations, though parametric data and tests are maintained at the highest level of indenture. They make use of Rational DOORS as their requirements repository, and use the Rational Gateway functionality to relate the model and parametric data to the requirements so they can track how well they are doing in terms of traceability and satisfying specific requirements as well as the requirements as a whole (Figs. 12, 13).

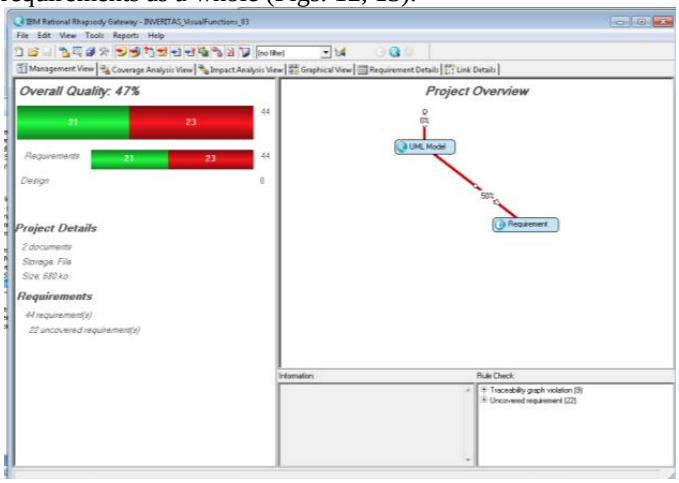

Figure 12 - Rhapsody Gateway, Requirements Management View [10]

These tools are the outgrowth of the concept of including parametric and performance data in requirements and being able to model those requirements and the tests done to validate them as part of the system model. Each requirement is verified by what subsystem or block satisfies it and what tests have validated that the requirement has been met. Changes made to the value of a requirement in a low level block can be tracked up to the highest system level to see what impact changes have on the viability and functionality of the system as a whole.

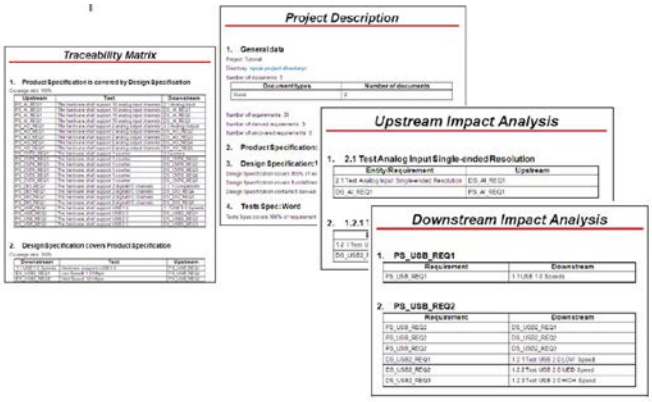

Figure 13 - Requirement Traceability Report [10]

Scenarios are a set of descriptions of each interaction that function might facilitate, with a dynamic view of the interaction between all blocks at that level. This expands on the architectural design, which shows all the blocks which make up that function/subsystem, the interfaces between them, and their dynamic behavior [9] (example part in Fig. 14).

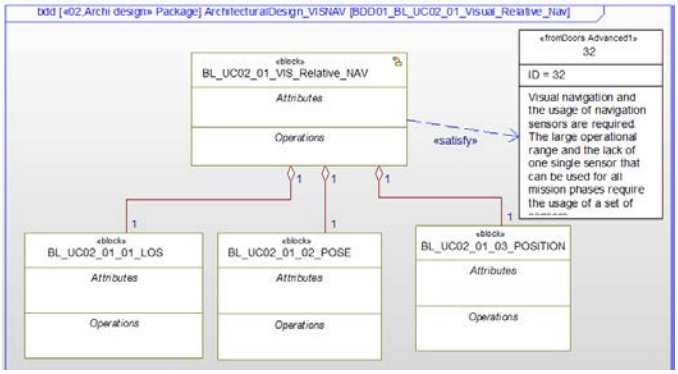

Figure 14 – Sample of INVERITAS Architectural Design Blocks as a Block Definition Diagram (BDD) [10]

One of the diagrams that sets OOSEM apart is the Internal Block Diagram (IBD) (the following case had a clearer IBD structure and is included here in Fig. 15). The IBD opens up a functional block and decomposes it, showing interfaces with other blocks as declared physical data ports. This allows for generating views in which it's possible to see data flows between blocks without having to specify them in manually created diagrams, and for changing how data flows in one diagram and having it propagate to others that relate to it. [10]

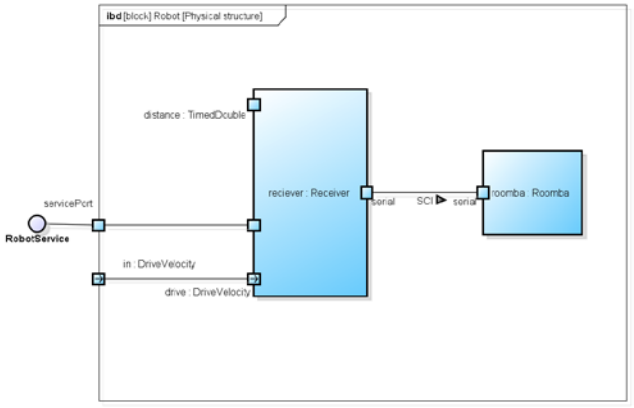

Figure 15 - Sample IBD [11]

They also made use of state machine and sequence diagrams (not pictured, effectively the same implementation as in UML based frameworks), to show how messages and processes flow, and how they change the state of the system, which was the lowest level they reached during the project as covered by the two papers they wrote. As of the papers, they had not done a second or third level functional view. Their conclusion was that SysML would require a strict definition of guidelines for space based project work before it could be rolled out to the rest of the European Space Agency. The INVERITAS project ended in 2012, having successfully shown a terrestrial prototype of the rendezvous and capture system.

C. SUPPORTING CASES – SYSML

While the INVERITAS authors provided quite a lot of useful material, they did not use all the tools available and did not provide examples of all their activities. A case provided by Kenji Hiranabe, at Change Vision, Inc. shows us several more activities and diagrams involved in the process through a project on creating a system to control multiple robots using whole body gesture interpretation [11]. They used a Kinect to control a group of Roombas through a PC. They defined their requirements by a set of core requirements (ie. Mission) on one diagram and specific system requirements on another, which were then connected to blocks in the architecture to show that they were being satisfied, and were connected to derived requirements based on the core requirements or system requirements (an example of each kind is contained on their Robot Requirements system requirement diagram, Fig. 16).

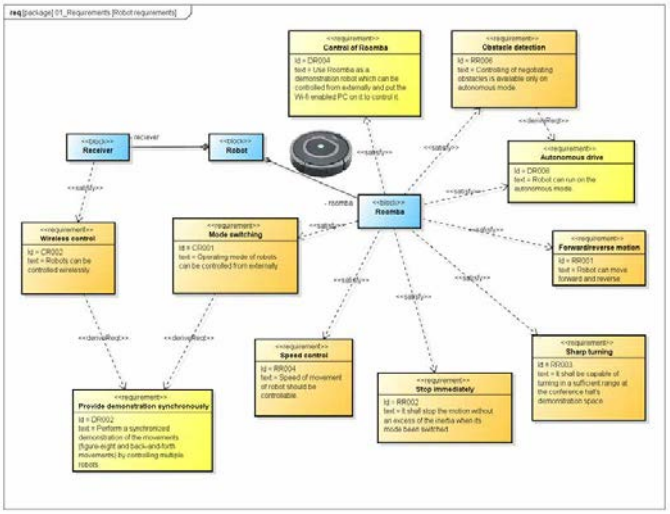

Figure 16 - Robot Requirements [11]

In addition, their use cases included blocks which were the highest level systems that were being executed by the functions in the use case so as to better connect all the elements together. Their context diagrams included actors for the same reason, helping to illustrate that autonomous systems are influenced by their environment even as they execute their tasks (Fig. 17).

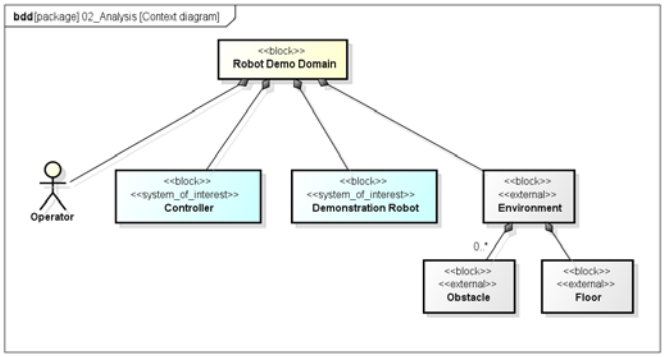

Figure 17 - Context Diagram [11]

In these diagrams and activities, we can see more clearly the usefulness in early stages of OOSEM and SysML in general. By combining actors with blocks, and requirements with simulations or testing, SysML keeps the process closely linked in all phases. OOSEM specifically also keeps the organization-level needs in mind, so that at every stage of development there is no loss of clarity in requirements, user needs, and organizational advantages when decisions are made.

IV. ANALYSIS

The COMET method makes a compelling case for itself to the often code-driven world of advanced robotics. It's short, produces easy to read artifacts, and models both the static and dynamic aspect of a system that reacts to its environment in real time and often without significant user guidance. It focuses on the key aspects of development. This is what you want it to do (requirements), this is what it's made of (static), this is how the parts react in motion (dynamic), and this is how it all fits together (consolidated collaboration). It gets to the point and provides nothing more than is needed to express a system to be built, which is what many rapid prototyping projects desire. They want a skeleton so they can hang whatever interesting technology they have on it and see how all the bits play together, knowing that the core concept of its function and design is well expressed and understood.

The biggest drawback from a project standpoint is scalability. COMET is the beginning of a framework, but it does not carry on from early design to implementation, traceability, maintenance, etc. It expresses the concept and lets the developer work out the details without keeping track of whether those details work. It also lacks testability, and enterprise level support.

The biggest drawback technologically is that UML is a software standard and while COMET might work for the software side of an autonomous system, it does not change the fact that most robotics projects need to express their hardware interaction as more than an idealized reaction to a data driven stimulus. UML ends at the software interface and hardware is an abstract external device which is *operated* by but not *defined* by the system modeled. The T-Rot developers did their best to work around this fact, but the cumbersome way they had to express themselves showed that it was becoming a problem. One example was that state changes had to pass through a central node in the concept, meaning the robot will execute what is effectively a state change in one part, but not 'know' it conceptually until message traffic has reached some central state processor. This could become a problem later in development and lead to sub-optimal implementation of the system using logic with makes sense in the context of the COMET framework but not for good design. In addition, most of their hardware was already defined or COTS, and so while there were still hardware performance requirements for things like sensors, the core of their project was pre-defined and so out of scope. The hardware was more often a constraint than a design decision.

For the purposes of a project like T-Rot, COMET UML does the job. It gives a distributed team a common reference

with which to plan data flow between separately developed subsystems which are built on mainly COTS or already provided hardware items. In a rapid prototyping environment with predefined hardware, COMET works very well as it keeps the team focused on development rather than planning, giving them only as much information as they need to make sure that the entire distributed team knows that the others are doing and how their piece fits in. Internally if T-Rot becomes more than a demo piece, the individual teams may need more than COMET provides even if the project as a whole continues to use COMET to maintain a common full system concept.

OOSEM with INVERITAS and the supporting case show a more structured and holistic view. Neither case used the enterprise level capabilities of the framework – which makes sense because in neither case was there an enterprise level organization involved. Focusing on the parts they used, it is readily apparent that for a larger scale project in which major hardware or even the platform can be designed or specified by the project that OOSEM SysML is a very effective tool. It is structured, goes from requirements to testing, and defines aspects of the system on the parametric level. Tests can be included to show that requirements have been satisfied, so that almost all aspects of the project design can be tracked in the model.

By defining hardware as blocks alongside software, an autonomous system can be readily built in the model and perhaps even simulated in the same model as it is designed. The issues with this are mainly those of the computation and upkeep of a simulation in-model. In the case of INVERITAS, this is not likely to occur as their models were very complex. The ESA did adopt OOSEM as their preferred framework, and looked into the concept of simulating in-model – which would have maintained the model-driven approach even further into their process – but determined that as of now the tools were unsuitable for running simulations that involved as many variables as occurred in a space-based platform [12]. They favored a distributed approach in which simulations were done in their own software and on computers capable of simulating real-time orbital systems (Fig. 18).

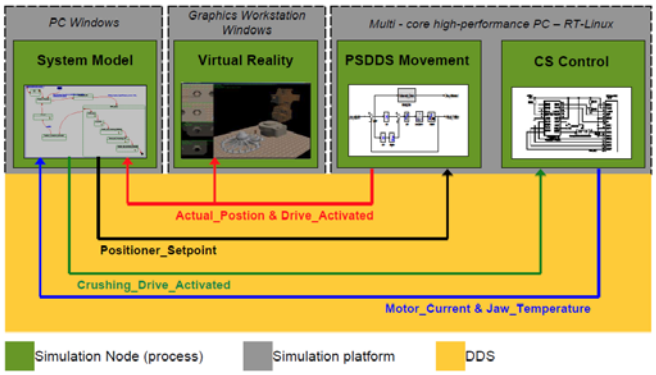

Figure 18 - ESA Simulation to Model Linkage [12]

The main drawback to OOSEM with SysML is that it tends to require tailoring before it is used in any particular application. This is partly because –as INVERITAS found – SysML is not a highly structured language. The difference in format between INVERITAS and the gesture control project in the supporting case shows that unlike process frameworks with highly defined artifacts like ACDM, OOSEM's implementation of artifacts in SysML is up to the individual project to define. Both the ESA's examples (one of which is Fig. 18), and the gesture controlled Roomba show much greater clarity than IVERITAS through the simple addition of colored blocks and pictures. That said, once a styleguide is produced, this is no longer an issue from an enterprise perspective and arguably phase 1 of OOSEM specifies that this is exactly what is supposed to happen. A framework that is not flexible is less likely to survive changes in industry, and when it is dependent on a modeling language that also changes with time, there is no way OOSEM could be permanently defined with a specific set of actions and artifacts.

As the ESA observed, there is also a lack of simulation quality in current system modeling software. This limits simulation to small-to-medium sized systems, which may be the best that can be expected. There is only so much a single piece of software can do, and the ESA found that they were able to integrate their external simulation software into the model software through a port. The model only needs to know the results, so though a very powerful simulator would be useful since that allows for improving any code that it might generate, it is not necessary for a model-driven framework.

For INVERITAS and the supporting case, OOSEM worked very well for their framework. It provides a means of putting together most of the process and keeping track of it in a way that makes sure that changes in one place are reflected in the entire system and in the requirements. Though the European Space Agency might need more structure for a project as critical and regulated as an autonomous spacecraft operating in the same region as extremely valuable communications and scientific satellites, OOSEM at least provides the backbone of a design process that keeps the project somewhat flexible and well placed to make changes or track requirements and design decisions to their source as well as integrating testing with the rest of the design process. Further artifacts will no doubt be necessary to satisfy regulatory agencies as well as the stringent needs of the ESA, but for internal development and perhaps even some external document generation, OOSEM appears to work quite well.

What neither framework provides in terms of project needs are risk management, comprehensive maintenance strategies, or means of tracking real deployment – though COMET as part of UML does show the deployment of data, that is not the deployment of the *system*, merely how data will move once the system is operational. As such, neither can be considered as complete process frameworks and would need outside artifacts and processes in order to be used for a project that encompasses the entire life cycle. However, the enterprise level nature of OOSEM adds features that are usually lacking in even the more mature process frameworks, looking at

organizational impact of the project and how each component might be useful in a library of software and hardware solutions.

For the specific needs of robotics platforms, the most important thing that is missing from either modeling language in the examples shown is an easy way to show autonomy. Most sequence diagrams assume that the process will reverse in direction due to errors or mistakes by the operator, whereas autonomous systems often backtrack as part of an algorithm or internal process. In these examples, I did not see a sequence diagram that appeared to take that into account. This is in fact very tricky, as many software solutions do have loops, but usually those loops are either parts of state changes or far enough into implementation not to need modeling. Here, there may be a single command given and then everything else is a series of environmental stimuli that can have combinations that are difficult to predict. As the autonomy division at Google has learned, the operating environment can get very strange and occasionally the system will have to decide what to do when it sees a woman in an electric wheelchair chasing a duck back and forth across the street [13].

V. Conclusion

Modeling is still not yet a major activity within most robotics development. The tools are there, however, and of those tools COMET and OOSEM both fill good niches in developing autonomous or mechatronic solutions. I personally favor OOSEM in this, as of the two it is closer to complete. Additionally, as has been shown in this analysis, UML lacks many capabilities provided by SysML which make for the holistic solution needed for merging hardware and software development.

As robotics solutions become more prevalent, hardware and software will need to be modeled in tandem. The automotive industry has adopted SysML processes, and I believe that SysML will be what most large projects will prefer. It is closer to how mechatronic systems are developed conceptually, and in the near future I expect a version of OOSEM for robotics to be developed.

Acknowledgment

The author thanks David Root and Eduardo Miranda for their excellent course and Carnegie-Mellon University for creating a place where this kind of cross-program collaboration can occur, with the hope that cross-college collaboration will increase to where the MBA and MSE/MRSD programs can freely send students to learn the art in full.

References

[1] J. a. C. H. I. Huckaby, "A Case for SysML in Robotics," in *2014 IEEE International Conference on Automation Science and Engineering (CASE)*, Taipei, Taiwan, 2014.

[2] Defense Acquisition University, "Acquisition Process - Defense Acquisition System [DAP]," [Online]. Available: https://dap.dau.mil/aphome/das/Pages/Business.aspx. [Accessed 2 August 2016].

[3] "UML 2.2 Diagrams Overview," 2009. [Online]. Available: http://www.uml-diagrams.org/uml-22-diagrams.html. [Accessed 20 July 2016].

[4] S. Friedenthal, A. Moore and R. Steiner, *OMG Systems Modeling Language Tutorial*, INCOSE, 2006.

[5] H. Gomaa, "Designing Real-Time Applications with the COMET/UML Method," Fairfax, Virginia, 2000.

[6] L. E. Hart, "Introduction To Model-Based System Engineering (MBSE) and SysML," in *Delaware Valley INCOSE Chapter Meeting*, Delaware Valley, 2015.

[7] J. A. Estefan, "Survey of Model-Based Systems Engineering (MBSE) Methodologies," Jet Propulsion Laboratory, Pasadena, California, U.S.A., 2008.

[8] M. Kim, S. Kim, S. Park, M.-T. Choi, M. Kim and H. Gom, "UML-Based Service Robot Software Development: A Case Study," in *Proceedings of the 28th International Conference on Software Engineering*, Shanghai, China, 2008.

[9] S. Chhaniyara, C. Saaj, M. Althoff-Kotzias, I. Ahrns and B. Maediger, "MODEL BASED SYSTEM ENGINEERING FOR SPACE ROBOTICS SYSTEMS," in *Proceedings of ESA ASTRA*, Noordwijk, The Netherlands, 2011.

[10] S. Chhaniyara and C. Saaj, "SysML BASED SYSTEM ENGINEERING: A CASE STUDY FOR SPACE ROBOTICS SYSTEMS," in *62nd International Astronautical Congress*, Cape Town, SA, 2010.

[11] K. Hiranabe and N. Ando, "Using SysML in a RTC-based Robotics Application : a case study with a demo," in *Robotics Information Day at the OMG technical meeting*, Burlingame, CA, 2012.

[12] T. Krueger, *Modelling of a Complex System using SysML in a Model Based Design Approach*, Automation and Robotics Section, European Space Agency, 2011.

[13] D. Muoio, "People do some truly crazy stuff when they encounter Google's driverless cars," *Tech Insider*, 5 April 2016.

Paul M. Calhoun (M'10) is a graduate student in the Master's of Robotic System Development (MRSD) program at Carnegie-Mellon University. He graduated with a dual degree in Electrical Engineering and Operations Management from Stony Brook University, after which he spent three years working for the US Navy as a project lead in the SEWIP program. He is a published science fiction author, a panelist and moderator at several science fiction and anthropomorphics conventions, and seeks to hasten the dawn of the transhuman era. As part of this effort, he is working on human robot interaction and integration, having recently finished teaching a Baxter to dance.