Three Reasons why Document-based SE (usually) works better than (most of) MBSE

Jean-Philippe LERAT

SODIUS sas – 1 rue André Gide – 75015 – PARIS

jplerat@sodius.com

Abstract. Under this provocative title, this paper warns the reader against a common belief: MBSE is the key to success. This statement is not wrong but terribly reductionist since the success of MBSE requires good SE practices to begin with. However in real life, most of MBSE efforts are just reflecting the bad SE practices that it pretends to correct. This mirroring does not help to solve the problems nor to implement better SE practices. In general, it just bastardizes what was already hardly working, based on document exchanges.

As a way to support this demonstration, the paper highlights a number of problems that, if not addressed, ruins any MBSE implementation.

Successful implementation of MBSE is a crucial factor of survivability for most of companies and agencies in today's world. This silver-bullet is, in many cases, the last ammunition of the projet managers: that's why it is worth to pay a bit of attention to the target, before shooting.

Where do we start from?

Blindness in SE readiness level In 2003, for the INCOSE Symposium held in Washington DC, I did present a paper named "Will Car Manufacturers Survive the Second Death of Henry Ford?". To make it brief, this paper was pointing out how the architecture of a vehicle was still inherited from the genius of Henry Ford. Unfortunately, after 80 years of success, the architecture was close to collapse, causing more troubles today than solving problems. Making a parallel between "fly-by-wire" and "drive-by-wire", the paper was ending by a simple statement: "be careful: automotive industry is unknowingly but surely collapsing". In the room, some people did look interested, while most of the audience was smiling.

Under-engineered and over-managed projects. In the early 70's, the mankind was fascinated by the success of technologies: moon walking, earth exploration, generalization of nuclear power, sophisticated weapons and counter-measures, subaquatic successes, promises of genetic sciences for new medicines and better-than-genuine vegetables… Lead by the US/USSR race and fuelled by endless funding, technology was pushing the limits everywhere, promising the future to look bright for everyone. In Western countries, many people were worried with "how to spend time in

2020, when the need for labor will have disappeared?". Unfortunately, delusion was cruel for all of us, when the '80 and '90 did show that these objectives were not reached.

As projects were all late and spending much more than their allocated budgets, the mood was to consider that engineers had no ideas about deadlines and funding. Therefore, the power was transferred from engineers to managers, who raised the level of awareness to time and money. However, despite this necessary shift, the assumption was still that "Yes, we can [do everything with technology]". The western countries, so proud of its domination of nature by high-technologies, did blind themselves to a critical point: we still have technological problems, we still are bouncing on limits in knowledge, and we still face unknowns and uncertainty.

The place to address those issues is the art of architecting systems. But, overconfidence in technical capabilities leads to reduce the importance of architecture, which is usually the poor-parent of the SE family.

Collection of Model-Based activities rather than Model-Based Engineering.

A very common assumption that can be heard all over INCOSE's members is "we do MBSE, because all our activities are based on models". However, such a statement is a dangerous syllogism. In the last thirty years, modeling is used in all scientific and technological fields. Today, every discipline has its modeling skills, with a number of solution providers. Mechanical engineers draw virtual mock-ups; planning people use plenty of COTS for synthesizing time and resources consumption; electrical engineers let modeling software calculating wiring sections and thermal dissipations. Obviously, the software discipline comes with a wide array of UML and non-UML modeling capabilities.

So, manual activities in large Systems are now rare, which gives the impression that projects are following a model-based process.

This is very often not the case, because MBSE is different from a patchwork of models issued by separate activities.

It must be remembered that Powerpoint is still the most popular architecture tool.

The situation is sometimes getting even worse with Architecture Frameworks like DoDAF or NAF. These xAF are sets of diagrams required by Acquisition Agencies. These models are end-products and do not reflect the design process that was used to produce them.

As a result, most of companies compile these views as a deliverable for the customer, rather than as a guide for the architects.

Putting SE back on the right tracks. So, after this rather boring and pessimistic point of view, is there still room for hope and constructive criticism? Yes, and plenty!

Today's level of modeling in each activity like mechanical, human factor, software or electrical behavior is a key asset to success. But even if it is necessary, it is not sufficient.

A MBSE process is not just a collection of MD activities !

Despite the declaration of most of project managers, what is missing the most in many systems is an architecture, which is the description of principles that are ruling the system: principles ruling its behavior, principles ruling its organization, principles ruling its interaction with its extended context of operation.

Simplistic statements?

Unfortunately, no... A lot of systems are currently under development, whose principles for their future life are not defined or poorly demonstrated, even when tons of models are produced. Worse: many systems are not architected, even if many modeling tools are used to represent their “architecture” but rather display sets of products.

Where does MBSE brings confusion here?

The case is not rare where SysML diagrams, xAF views or UML models are used as presentation means, without any connection with the reality of the future system. This leads to the impression that the system has a strong architecture, while it just hides the fact that the system does not have any.

In these not-so-rare cases, it’s better to keep the old Document-based SE process: it’s old fashion, outrageously expensive but, by the end, it happens to work. On the contrary, such a bad MBSE, hiding a lack of architecture, is a way to quickly lose a lot of time and money for no result at all.

From a personal experience, here are three main pitfalls that support this vision:

- Poor MBSE focuses on Product Modeling rather than on System Engineering
- Poor MBSE generates complexity rather than reducing it
- Poor MBSE does not help to share information

First Reason :
Poor MBSE focus on Product Modeling rather than on Systems Engineering

Back to the basics, SE is decomposed of two words: one being “Systems” and the other one being “Engineering”. The definition of the last word is very challenging: doing systems – either as a company or as an individual – is not a sufficient condition to be considered as being SE-capable – still either as a company or as an individual. Engineering means to follow a process that optimizes the delivery of a working system, able to perform its intended mission. Many great systems exist that have never been engineered: they came from a trial-and-error based process, which is the usual one.

The ability to really "engineer" systems is given by a real and accurate capacity to imagine and predict the future. Relevant and precise information about the anticipated behavior of the system is the only way to escape from a process based on Faith and Hope.

A Technological Divination Process? Despite not being named as such in any ISO standard, this process transforms an intellectual vision of a problem's solution into a set of artifacts on which engineers can work, simulate and predict performances. It mandates the team on duty to imagine a solution for conducting the mission and, once validated, turn it into a concrete capability.
In this respect, systems processes are usually converging "loops", each helping to make the prediction about the system more and more accurate, in all of its aspects.

To achieve this, three domains of knowledge must be mastered:

- The Mission (what to do)
- The Environment (where and how the Mission will happen)
- The Consequences of Technical Choices (how my choices may alter, jeopardize or help the Mission and/or the stability of the Environment).

Unless these three domains of knowledge are confidently predictable, one cannot pretend to be doing Systems Engineering.

Another point: The main differentiator between "systems engineering efforts" and "trial and error-based progression" refers to the place of the architecture's validation.

True systems engineers validate the system's architecture by the end of the engineering, making sure that hazards in integration may impact some product's developments but not the principles that rule the system.

Model-Based Mission. The Mission is currently the heart of systems engineering. The SE process must provide the most accurate and precise description of the behavior expected by the stakeholder. The models must specify what the outcomes of the Mission are; who or what triggers behaviors and reactions from the system's products. Basically, it is a collection of use cases including "Nominal processes", "Alternate", "In training", "Under Validation", "Degraded", etc. and even "Deviant use" or "Under external aggression". Once these Use Cases are validated with its Stakeholder, then the System's Technical Requirements are elicited to shrink the different Use Cases into a minimal set.

Model-Based Environment. Under the name "Environment" is gathered all information related to the location and the circumstances in which the systems will operate. This includes meteorological, electrical or water supply, infrastructures of networks, availability of spares and consumable products, cultural issues with populations or users, operators or neighbors, thermal cycles, storages constraints, etc. Basically, it is a collection of Operational Situations, ranging from "Nominal", "Acceptable", "Exceptional" and obviously "Scary" or "Dangerous".

Model-Based Consequences. During the course of the design, most technical decisions may impact the system behaviors or the future operational environment. The most obvious example is a

radar: alone, it uses a EM-clean Environment, but as soon as two of them operates, they interact and fill the environment with waves that require special filters. The design of these filters is very sensitive because their specifications will be derived from a perturbation of the Environment known from the system's design rather than from any upfront analysis.

This point is certainly a key differentiator between SE and SwE processes. SE must feed the "Environmental Analysis Activity" back, and retrigger it, at each technical decision, even for small ones.

Model-based Validation. Validation is the activity that determines the extent to which the system will correctly handle the Mission. However, a system is not a physical entity: it is just a capability to conduct the Mission through a set of physical means. It is critical to obtain the validation of the system architecture before it is turned into concrete products. However, the architecture is not limited to the behavioral principles ruling the system, but also the relevance of the selected physical means to their intended purposes. This mean that the Technological Consequences of architectural choices are known and have been re-injected into the architectural processes. This particular task is a real challenge which mandates extremely predictable models.

Doc-based vs MBSE for System Architecture Definition

Information related from the three main domains of knowledge above, are the most reusable parts for any projects. They are also the most expense to acquire. Here stands precisely a major asset of large corporations: tons of data and models about the Mission, the Environment and Technical Consequences of Choices. However, this information is accessible through highly specialized simulators and modeling tools, created and validated through the years by highly specialized teams.

Most of document-based processes define activities around these simulations and early validation of architectures. It is easy to verify whether these numerous simulations were successfully conducted or not. This slow and time-consuming process gives opportunities to detect flaws in the candidate architectures.

Many of MBSE processes are disconnected from these simulators and believe that a bunch of Use Case diagrams is enough to specify the Mission in its future Environment.

In many cases, simplistic SysML diagrams are used to specify nominal cases at the very beginning, and the model is then frozen as the baseline.

This is not enough! Good SE practices model the nominal case first, then add all the extrinsic failure possibility handling, then add all the tailoring to the design decisions, then add all the intrinsic failure possibility handling, etc. Use Cases are enriched at every step of the process with more and more accurate data.

At each progression of the process, ALL the simulations must be conducted and re-conducted, since every step added information and refined the knowledge on the system's behavior.

A common pitfall is to consider the simulations handled by MBSE's tools as adequate, e.g. SysML modelers. It has been reported that some MBSE efforts were basing the validation of the system architecture on some simulations, ignoring the fostering of a multiple-simulated validation process.

Another pitfall regards the OO methods that are usually coming along with MBSE, specially in the SysML arena: Object-Oriented methods comes from the software world and describe systems as interactions of ***isolated*** "objects" exchanging messages triggering Methods, as shown below:

The objects are isolated because:

- The way they store data inside them is an internal affair of the Receiver and is not visible to the Sender.
- The way the object processes the message is an internal affair of the Receiver and is not visible to the Sender.

This is great since it enables changing the storing capabilities of objects and/or their processing implementation without changing the other Objects.

That works well for software, but the case of Systems Engineering is a bit different since the ways that data storage and processing is implemented is crucial for many non-functional reasons like safety, maintainability, security…

It does not mean that SysML diagrams cannot reproduce the interaction of the core of the objects, but it is certain that most of OO methods do not foster paying attention to these problems. Most objects are considered as isolated, even if Nature tells us that they are not.

Is Doc-based really better? The answer is obviously no… However, a doc-based SE shows many check points which are not used by most of MBSE. Model-based processes are usually jumping too fast to the implementation. Should a problem occur in the architecture, MBSE may not detect it as early in time as with a doc-based process.

Remember Johannes Kepler! Before he found that planets are ruled by elliptical orbits, the old modeling from Ptolemy was more accurate than the more-correct model of Copernicus. For sailors, Ptolemy's models were "better" than pre-Kepler Copernicus's models, which are technically "better".

We face the same paradox: doc-based SE is not as "correct" as a good MBSE, but it is safer than a bad one.

Second Reason:
Poor MBSE self-generates endogenous complexity

The Big Bad Complexity. "MBSE reduces complexity" is a buzz. Ninety percent of the presentations at this INCOSE Symposium are certainly starting by a sentence like "once upon a time, things were easy. But now, things are complex and thus, we need MBSE to tackle it".

Are we facing the right question?
From my experience, many MBSE processes that I have seen are increasing complexity rather than reducing it.

How? The answer is a bit abstract, but will certainly make the point:

At first, complexity is a measure of ignorance, especially in the behavior of systems. "complex" describes a system which is ruled by non-predictable laws. When the behavior of a system can be predicted through a formula, then it is said to be "complicated", however big the formula is (and by the way, sophisticated Swiss clocks providing lunar months, annual dating and solar tracking are named "complications").

As a joke, one could claim that a good SE process must turn a foreseen "complex system" into a "complicated solution".

To make it simple, a system ruled by a huge Truth Table of thousands of Booleans, it is not complex. Conversely, if a system reacts to only four [0…1] parameters, for which any variation, even at the sixth decimal, can unpredictably change its entire output, then it is.

Most of complexity comes from the course of the Mission, the volatility of the Environment and/or of properties of the Systems Physical Products.

A complex mission usually comes with multi-mission handling with automated reconfiguration. A complex environment may come from a wide spectrum of users or from physical uncertainties (for instance, on the Cassini-Huygens mission to Saturn and Titan, nobody knows whether the engine would land on a liquid or a concrete surface…).

Complex Physical Architectures comes from uncertainty in the individual behavior of the Products, unknowns on their failures causes and consequences, or unpredictable cocktails of physical or sociological phenomena, etc.

The System Hierarchy. Usually, Systems Engineering depicts three hierarchies of systems: One is the "System Hierarchy", another one is the "Functional Hierarchy" and the last one the "Physical Hierarchy". If the two last are easy to understand, the first one is often never explained, as if it was obvious. But it may be the most dubious one.

The best view point to understand the notion of Systems Hierarchy is to stand at the system's keystone: its Control-Command (C^2) principles. This C^2 is the synthesis of the main question of Systems Engineering: "How the Systems perceives its Environment and acts upon it to conduct the Mission"

Usually, systems C^2 strategy can be either:

- "raw" Analog, when a continuous signal directly perceived by a sensor is transformed into a suitable data using filters, intra- or extrapolations, capping, etc.
- Analog, where continuous signals are combined for getting Proportional, Integrated or Derived data from the "raw" sources, using computations, differential equations, etc.
- Booleans, where signals are transformed into true/false, zero/one information.
- Logical, where different Booleans are contributing to logical equations. Logical systems are typically controlled using Truth Tables, for instance.
- Probabilistic, where one or several members of the logical expression are not Boolean but a probability, leading to transform the logical expression into a Bayesian net, for instance.

This list is not exhaustive and mathematicians will likely add many other types of C^2 principles. However, what should be understood is that system engineers face many C^2 strategies and must select the most appropriate.

Here stands one of the most dubious pitfalls in Systems Engineering: each system and sub-system has its own working principle, each having its own C^2 keystone. The C^2 principles of the system, is different than the ones of its sub-systems, which differs equally from its assemblies.

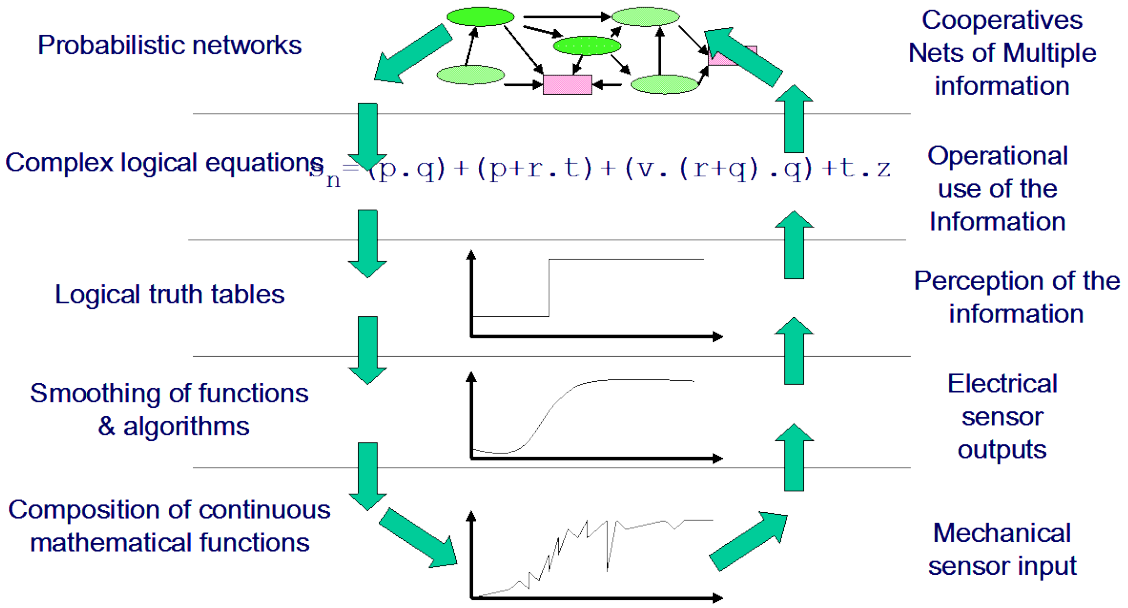
For instance, here are the different levels of C^2 in a missile:

- The sensors of a flying missile perceive "raw" information like pressure or lights.
- The target-seeker is analog and computes several sensor readings to determine whether something is close or not, and it is if a decoy, a foe or a friend.
- The target-locking is logical and reacts when different conditions are met (seeker found something, it is in-distance, engine is still running, military payload in on, fire was authorized, no other target is already locked, etc.)
- The flight-conduction module is probabilistic and configures the navigation and military effects depending on the best probability to hit a suitable target amongst the various one that are detected.

- The missile-system is probabilistic at another level, and depends on the external Environment (war/peace time, operation/exercise, polar conditions/sand desert…). The mission is to deliver a military capability on a war theater.

These subsystems do not mix together, while they are sharing the same data: a photovoltaic sensor gives raw data of a light, which is then transformed into a photography and spectroscopic information, and then it is transformed in a digital signature, which will trigger to a potential target-locking process, which enters into the mission configuration.

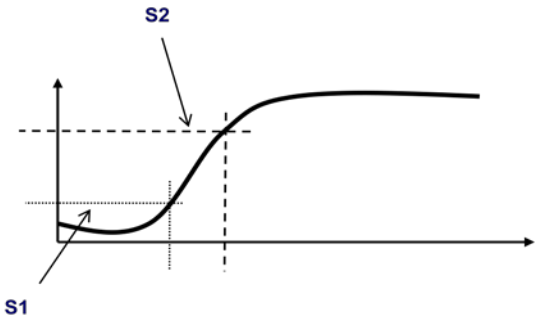
At each level, the information given by the "next level" must be used by the C^2 engine in a suitable form, either "upward" or "downward".



The hierarchy of systems from their control-command relationships

One of the problems that remains in SE is precisely the transformations of information within the system hierarchy.

Considering the transformation of an Analog signal into a Boolean, the System Engineer defining the C^2 engine for a system's level faces problems like below:



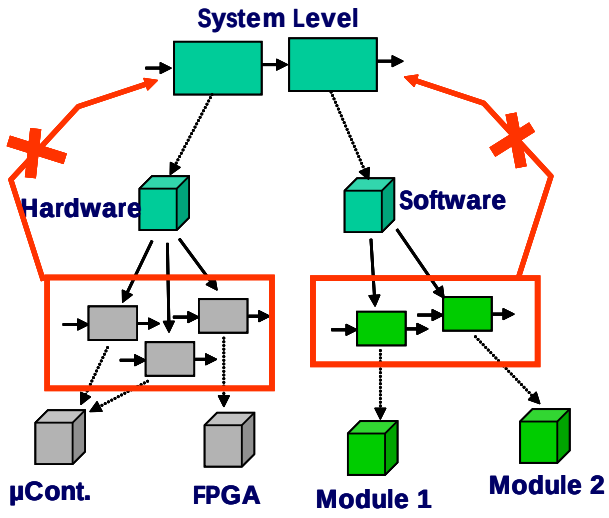
Is the S1 strategy better than the S2 one, since the system "at this level" will be more reactive, but may be less reliable in case of a glitch signal?

This transformation of continuous signals into Boolean ones is certainly one of the worse that we face in Systems Engineering: most of the signals are analog, while most of the models are based on state machines.

The rupture between systems levels. The border between the system and its subsystems is usually made by the rupture between two different C^2 principles.

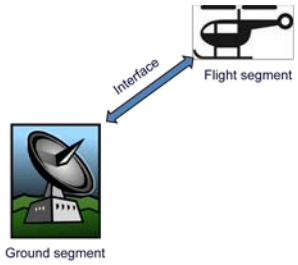
However, decompositions do not "cross the ruptures". This mean that between two levels of systems, ruled by different C^2 principles, no continuity can be assumed, nothing may be "carried over".

In simple words: functions of a sub-system are not sub-functions of the system.



Functions of sub-systems are not sub-functions of the system!

Another consequence is that Interfaces appear and disappear at various systems' levels. An interaction may exist between high level components and may not exist when they are considered as systems themselves, as shown below:



The tactical interaction between the helicopter and the ground station exist, and the transmit/receive functions are allocated to each segment. But when each end is then created (for instance the helicopter) then its internal interfaces of the radio are beams, EM, electrical and mechanicals. By the end, the communication system handles all interfaces, but from a designer perspective, not all interfaces are relevant, or are even known.

Generating complexity. By experience, the lack of specification of the different C^2 levels leads to a potential high risk: stacking several levels of C^2.
The way to reach the exact opposite of the goals is very easy and is just the consequence of this stack of control-command laws, and is usually not detected by the usual processes.

Most of MBSE processes are just reflecting the use of a tool, which does not foster starting by "good questions" on C^2.

Amongst the "good questions" to ask, a bunch of them are directly linked to the principles: how to select the control-command principles that suit my system? This question is answered by several studies, some about the external factors (how the sensors perceives the environment) and some being related to internal factors (technologies, reliability, capabilities, availabilities…).

It is usual to observe that many projects do NOT really assess the suitability of their C2 principles to the system. As most of the C2 are software-coded, this assessment is usually subcontracted to the software pieces. However, IT engineers may not have the physicist background needed to understand the control-command laws, and all its theories.

Worse: many systems are not made from scratch but rather are evolutions of existing systems. In this case, the C2 principles of the existing system are almost never challenged against the new requirements. Unfortunately, in many cases, these new requirements are ruining the efficiency of the overall system because it introduces new needs for other principles.

For instance, many systems are conducted by state-machines and Truth Tables. A cyclic polling of parameters is made once a stable mode is reached, the values are discriminated, then a number of

capabilities are switched on or off., enabling or not a number of operation situations to be triggered or not.

However, if one evolution requests to speed up some acquisition process, or if some new sensors are added, or if some new sensors are less reliable than others, then it may occur that the polling of all sensors becomes impossible at the sampling frequency. At this time, an uncertain number of sensors produce uncertain values: the system cannot be managed any more by its Boolean logic but must be ruled by probabilistic solutions, such as causal networks or Bayesians nets.

What happens if the system engineer does not pay attention to that border? Then the system is ruled by the wrong principles.

In the worst case, the system will simply never reach its objectives. In the best case, the system is fully jeopardized and is likely to generate a lot of unexpected reworking.

Several times, this failure to check for the right control-command engine has been seen, and even at several levels. For all of these mistakes "sunk" into components, the last parts are usually extremely difficult and complex, leading to costly and sensitive software components.

Several C2 software components controlling large systems are made more complex by design rather than by nature. If one must code an analog device to run large Boolean equations combining probabilistic values, then it becomes much more complex (and usually less efficient) than these three elements separated and implemented in a smart way.

Doc-based vs MBSE at System Structuring

Document-based processes are less likely than MBSE to stumble on that pitfall.

By definition, documents are held by responsible persons. In this respect, each stakeholder tries to protect its liability domain: the architect avoids being accountable for the software, while the EDA folks want to keep away from problems raised by mechanical structures.

Twenty years ago, system-level modeling tools fostered by the "TRW methodologists" brought to the market products like DCDS, RDD-100 or CORE, which all were driven by a simple statement: only three views are required for a "system-block" (behavioral – physical breakdown – interfaces/N^2). However, each system-block was requiring its own three views. Strict configuration and interface management was mandated to keep all the levels synchronized and consistent. This was the beginning of system configuration management.

These tools, reflecting document-based processes, were maybe a bit difficult to use but were fostering the notion of "system hierarchies" which is reflected by ISO15288 in "Systems blocks", as composition of non-mixable C^2.

Today's modeling tools are working on another paradigm: they try to embed all system levels within a single, huge model, making an extensive use of the refinement techniques.

Unfortunately, hooking classes on a 10m x 10m diagram like ornaments on a Christmas Tree does not work: many of the systems components do not address the same level of C^2, making them irrelevant to each other and thus, unusable for simulation or whatever else.

Why? Because: refinement of functions cannot be done forever and must respect the C2 principles, unless the architecture is ruined.

Again, is a Doc-based process really better? Of course, no… except that document-based processes slow down the architecture design so much that it helps to spot that pitfall before falling in it. Most of MBSE processes will dump models and model goodies into a large database, spending millions before someone understands that it is unusable and will be too bloodily complex to implement. Worse: once the model is too big, then its weight rules the entire project, not its technical relevance.

Third reason:
The System Modeling Legends of information sharing

Models, Models, Models Everywhere!

The first pitfall that we did raise was connected to the lack of good modeling practices in most enterprises.
The second pitfall is related to a lack of good systems organization in most of programs.

This third pitfall regards the lack of good technical management in most projects.

In many cases, the insertion of MBSE does not come from a system-vision, but is raised by the promotion of a tool from the most important technical discipline as "the systems engineering tool". Depending on the kind of system, that predominant tool can be either a former CAD/CAM tool, a software tool or even a planning tool.
When facing that case, users usually do not separate the system-level information from the implementation of its main parts.

By doing that, user blurs the border between the C2 of the system with the C2 of the part. This reproduces the second pitfall but within the implementation of end-products.

When mechanical engineering rules, the entire Systems Engineering is usually limited to the production of a Bill of Material (BoM) and its efforts is on the geometric allocation of volumes and weights, which is only a fraction of SE.

When software engineering rules, the entire Systems Engineering converges to information management between the objects and the IT system to support the exchanges. This is crucial, but this is only a portion of SE.

However, when a tool is promoted like this, it usually becomes an “attractor” for other disciplines. In the CAD/CAM arena for instance, the tool started to be used to define electrical cable space allocation. Then the tool stretched to define wire specification. Then the tool attracted electrical engineering tools to define wire specification, thermal dissipation, wire section, etc.

In the software arena, the same approach can be seen with the use of UML in all parts of software parts engineering: code, networks, messages, databases, etc. Tomorrow, the EDA people will use also UML to define their FPGA, while physicians will use SysML diagrams to specify mathematical equations through the new “Parametric diagrams”.

None of that is “bad”, but it leads to inflated models with a huge quantity of implementation data, rather than to separate the system-level and the different implementation levels.

By the end, most of these diagrams are fat, unusable, controlled by the implementation disciplines.

Smashing the architecture. Considering how little place is made for architectural efforts in projects, there is a big risk that a SysML tool providing profiles for each implementation discipline will hide completely the architecture. Architecture must be the glue between the disciplines, and not just an extraction of implementation data.

Interoperability problems. To enable Universality in modeling, UML has great features like a unified notation to create profiles, which handles the precise semantic of models. However, theses profiles are by nature incompatible between each other. There is a big risk to believe that all UML-speaking models can be easily mixed together.

External silos made internal. As a result of a unified modeling tool (either coming from CAD/CAM or UML/SysML) generalized to every technical discipline, the companies partitioning, based on separate and non-communicant departments, currently having their sanctuaries in their own tools, will remain: the sanctuaries will remain in UML packages instead. People that do not communicate today because of documents will not communicate tomorrow because of models, increasing unified autism rather than unified endeavors.

It’s time for a serious joke:

In a spacecraft, where stands the SysML port between the “system” and vacuum?

Doc-based vs MBSE at Information sharing

Today, most of the information is not shared, whatever the IT/PLM manager says. What most of IT/PLM manager refers to is a vault in which every model is stored and accessible. Sharing information is far from just making it accessible.

In a document, the information is stored and gathered in a logical sequence. People are expected to understand the information and find it easily through glossaries and index (and especially documents that have been digitalized, scanned, and indexed by the PLM solution). A document is also something

that can be exchanged and annotated at the coffee machine or while being in a train. It is also easy to see when a document becomes unusable: too heavy, too complex, largely redundant, etc. Many systems were rearchitected because their documents had to be re-architected by good system engineers.

In this regard, MBSE fosters the idea of exchanging models. How does it work? For instance, once released by its author, the information is available through viewers (but who knows how to deal with 10 viewers for CAD/CAM, UML, EDA, performance, planning, requirements… ?). However, most MBSE tools do not include any quality measurements: who can handle a UML model with 3,000,000 classes? Who can draw any information from a Statemate model made of three charts and 50,000 lines of behavioral C? Who can share a Matlab model of thousands of uncomment blocks?

Of course, they are not "shared", but just "made available".

Garbage in ➔ Garbage out…

As most of models made are not usable, sharing them does not solve anything.

So, is Doc-based really better ? Again, the doc-based processes are slower and less efficient than a good MD-based process, but much less risky than a bad MD-based one.

Succeeding with MBSE : AR-CHI-TEC-TING !

The usual statement for MBSE is that it implements good SE efforts.

It is worth it to challenge that statement. Personally, all successful uses of MBSE were in projects already having a true system vision and a good doc-based process. For those virtuous users, the introduction of tools has been efficient and cost effective, because tools were given to efficient engineers dealing with a cost-effective context.

MBSE never saved anyone from a failing doc-based process.

Before populating any MBSE tool's database, it is good to survey the real readiness of the teams with a couple of simple questions:

- Are we committed to make architectural work? (If your answer is no, nothing will redeem you from failure – MBSE or Doc-based).
- Are we committed to have clean borders of our systems C^2? (if your answer is no, May the Providence preserve you from crossing a C2 rupture!).
- Are we committed to facilitate communication in the team, and not only the gathering of separated models? (if your answer is no, then you can already search for extra time and extra budgets)

If you answer “no” to any these three questions, then the likelihood of failure of your MBSE process is very high… It may be time to keep your documentation-based process and find a way to improve your readiness in implementing MBSE.

Remember: good MBSE starts with humility and skepticism, two basic assets of any engineer, and especially of systems engineers.

BIOGRAPHY

Jean-Philippe Lerat is the founder and CEO of SODIUS, a company specialized in Systems Engineering, also recognized as a leading supplier of interoperability tools for large corporations and OEM partners. His background started in firmware and software, while his first experience with a “system vision” came from Apple Computer, for which he worked in the 80’s. Since then, he worked for various startups helping large companies in multidisciplinary projects in Space, Automotive, Defense, etc.
Jean-Philippe is one of the oldest veterans of INCOSE in Europe (since 1998), and a proud member of AFIS, the French chapter. He is also active since 1996 in several standardization efforts at ISO and IEEE. Jean-Philippe Lerat is currently an INCOSE Ambassador – INCOSE Liaison for ISO SC7 – He is also a board member of the “Pôle de Compétitivité Images et Réseaux” – a cluster for R&D and prospective visions in the field of public R&D funding for media and networks.