Chapter 1

Model-Based System Architecting and Decision-Making

Yaroslav Menshenin[1], Yaniv Mordecai[2], Edward F. Crawley[2], Bruce G. Cameron [2]

[1] Skolkovo Institute of Science and Technology, Moscow, Russia
[2] Massachusetts Institute of Technology, Cambridge MA, USA

"The straight line, a respectable optical illusion which ruins many a man."
— Victor Hugo, Les Misérables

Abstract. We explore the application of MBSE for conceptual system architecting. Choosing an architecture is a fundamental activity. Our Model-Based System Architecting (MBSA) framework facilitates the specification of an architecture as a reasoning process – a series of conceptualization and decision-making activities, backed-up by an MBSE environment. Our framework captures both the ontology of a stakeholder-driven and solution-oriented system architecture, and the process of growing the architecture as a series of conceptualization steps through five ontological domains: the stakeholder domain, the solution-neutral environment, the solution-specific environment, the integrated concept, and the concept of operations. Our MBSA approach shifts the modeling focus from recording to conceptualizing, exploring, decision-making, and innovating. In comparison to an "offline" architecting process, our approach may initially require a bigger effort but should enable stronger stakeholder engagement, clearer architectural decision point framing, quicker exploration, better long-term viability, and increased model robustness.

Keywords. Model-Based System Architecting, Model-Based Systems Engineering, Architectural Decision-Making, Object-Process Methodology, Concept Representation.

1.1. Introduction

Choosing an architecture for a complex system, sometimes called the "fuzzy front end" of design, is a task rife with ambiguity. Traditional approaches have relied on a federated mixture of informal, semiformal, and formal methods. The growing challenge systems face today has made these "offline" approaches largely obsolete. Model-Based Systems Engineering (MBSE) [1] is gradually becoming a mainstream approach for practicing systems engineering. However, while traditional systems engineering works to capture missing connections between subsystems, MBSE today is focused on the descriptive recording of concepts in models [2]. This concept representation is essential for further processing, analysis, and presentation, but it is only one aspect of systems engineering. Current practice and research are overweight with the representational effort in MBSE, and underweight on analysis and decision-making. Similarly, software engineers are expected to deliver operational, functional, secure, and efficient software regardless of the programming languages and software development environment they use; mechanical engineers are expected to deliver valid, verified, buildable, and maintainable part and component designs, regardless of the design technology they design with, etc. Nevertheless, in the current landscape of digital engineering [3], no one imagines that software, hardware, or mechanical engineers will not employ the latest software to manage, design, implement, test, and deploy their deliverables. Systems engineering should be no exception.

We explore the ways in which MBSE can be used to support system architecting, and to ensure that the process remains rigorous and insightful. Reaching a good system architecture must be inherent in any MBSE approach. Accordingly, our model-based system architecting (MBSA) approach uses models and analysis of MBSE to choose an architecture. It is not a detached adaptation or variation of MBSE to system architecting.

Much has been written about the descriptive aspects of MBSE, whether it be in cataloging functional flow or in defining potential system states. However, this documentation does not necessarily support architectural decision-making unless it presents decision points. A decision point could be an opportunity to choose a solution from at least two options. We define what we consider architectural, in order to evaluate how and where MBSE supports decision-making about architecture.

The effort involved in building an MBSE environment and the associated cultural transformation imply that the scope and purpose must be crisply

defined, so as to rationalize the investment in MBSE. One of these purposes (but by no means the only one) is to support architectural decision-making. MBSA includes the following cycle of activities in scope:

1. representing potential architectures with models,
2. identifying architectural decisions,
3. conducting analysis in support of emerging architectural decisions,
4. making architectural decisions based on model analysis results, and
5. capturing decisions in the model for the next architecting iteration.

Model-Based Conceptual Design (MBCD) resembles MBSA. MBCD is the application of MBSE to tradespace exploration during the conceptual stages of systems engineering [4]. The activities performed during the conceptual stages of system engineering are defined as architecting, and their main outcome is an architecture – a holistic view of the entire system. By contrast, activities performed to realize the architecture, particularly planning solutions with engineering and scientific knowhow – are considered as designing – where the main outcome is the design: a blueprint for developers to implement or build the system. A complex component's design may constitute architecting for that component as a bona fide system, e.g., the jet engine in an airplane, or a communication network that connects many sensors and controllers.

MBSA has also been used as an acronym for Model-Based System Architecture [5], in a framework which uses the Systems Modeling Language (SysML) [6]. That approach focused on providing a repository of artifacts, which facilitate communicating with stakeholders, assuring requirements traceability, and specifying systems and sub-systems. We employ the MBSE paradigm as a reasoning mechanism, and not only as a documentation approach, because we believe that it generates additional value to stakeholders.

Previously, the Model-Based System Architecting and Software Engineering (MBASE) approach [7] advocated a holistic process for software architectures, software lifecycle guidance. The MBASE approach was infact document-centric. The model-based ecosystems were not yet mature enough to accommodate a complete system architecting, design, development, deployment, and operation thread. Therefore, MBASE started with an Operational Concept Description (OCD), but focused on generating documents like the System and Software Requirements Definition (SSRD), System and Software Architecture Description (SSAD), Life Cycle Plan (LCP), and Feasibility Rationale Description (FRD) [8]. It also concerned some

critical aspects in software deployment such as iterative development, following the Spiral paradigm [9], transition, and software support.

The discussion on the necessity, relevance, and sufficiency of system architecting, particularly in software-intensive systems, has been ongoing especially with the appearance of short-cycle iterative and continuous development and deployment approaches [10]. The key argument remains, and gets validated in many famous failures [11] that holistic system architecting increases confidence in the ability to meet stakeholder needs, develop a robust architecture that can adapt to changes, and reduce the amount of technical debt as the system evolves [12].

Many of the building blocks described in this chapter originate from previous holistic frameworks for system architecting [13], in which modeling played a key role in concept description, but could not yet be regarded as a fully model-based approach.

Bahill and Madni introduce a model-based approach known as the SIMILAR process, which stands for: a) Stating the Problem, b) Investigating Alternatives, c) Modeling the System, d) Integrating Components, e) Launching the System, f) Assessing Performance, and g) Re-evaluating the System [14]. The MBSA approach that we proposed focuses and extends on the early stages in the SIMILAR framework and especially on early iterations in which the conceptual architecture is the main artifact, and little or no physical components are available.

1.1.1. Model-Based System Architecting: Crossing a Mental Grand Canyon

MBSA often begins with concept brainstorming in response to some need or set of needs, and ends with a formalized review and sign off of a well-defined and buildable architecture. In between, there is a series of conceptualizations and decisions: The leap from stakeholder needs to a well-defined organization of structural and behavioral elements does not happen overnight. This mental 'Grand Canyon' is simply too wide to jump all at once, and a series of intermediate steps is necessary. The question is: how can we wisely plan these steps that will lead us safely to the other side? This idea is illustrated in **Error! Reference source not found.**.

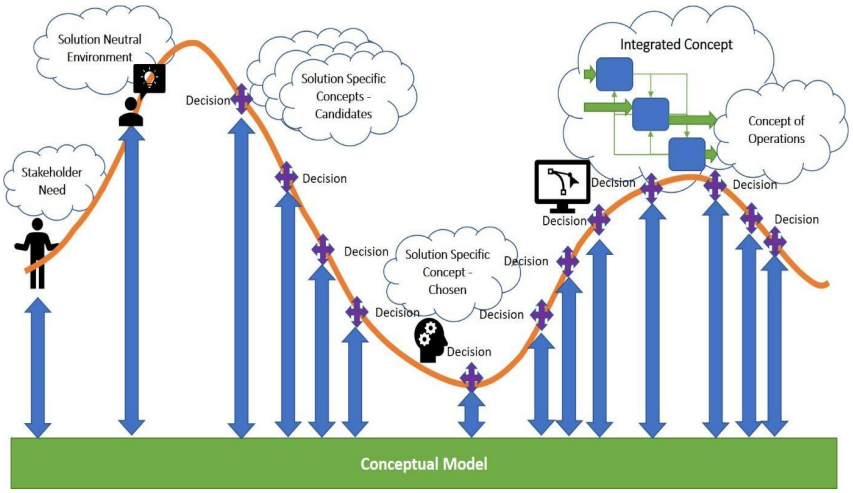


Figure 1. Crossing the mental Grand Canyon from needs to operationally-viable solutions, through a series of system architecture decisions, using a model as the knowledge base and primary reasoning engine.

A system architecture is a description of the structure and behavior of a system that jointly provide one or more functions to serve the needs of system stakeholders. MBSA relies on a formal modeling language to capture, present, and reason about the system architecture, but the deliverables are essentially the same as those of the traditional (not model-based) process: a specification of the system architecture, which can serve as the basis for further requirement specification, design, development, testing, and operation. This high-level concept of MBSA is illustrated in Figure 2. We shall be using Object-Process Methodology (OPM) [1] as a model for this Chapter, due its relative simplicity (using OPCloud, and its automatically generated text specifications[1] [15]). The complete reference model for our MBSA framework is included in [16]. In Figure 2 the objects (such as "Stakeholder" and "System Architect") are denoted by rectangles, whereas the process ("Model-Based System Architecting") is denoted by oval. The filled in black triangle inside a triangle means that the "Need" is the attribute of "Stakeholder". The link with arrow informs about the consumption of the attribute "Need" by the "Model-Based System Architecting" process. The link with filled in circle at the end is the agent link ("System Architect"), whereas the link with the open circle at the end is the instrument link ("Model-Based Systems Engineering Environment"). The full description of the OPM symbols can be found in [1].

1 Figure 2's title is directly drawn from the text specification that OPCloud generates for this diagram, making it an unambiguous description of the diagram.

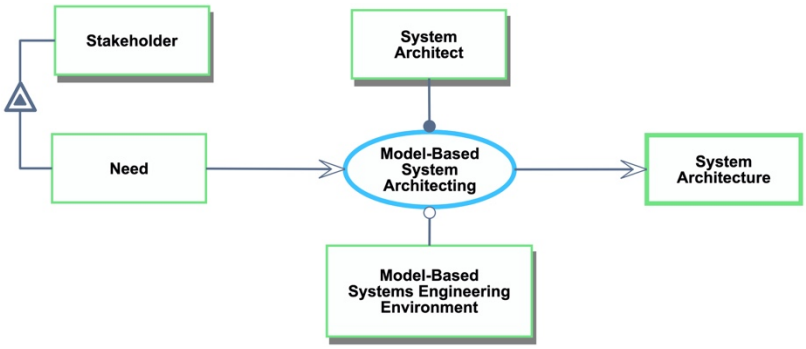


Figure 2. Model-Based System Architecting: Stakeholder exhibits Need. Model-Based System Architecting consumes Need of Stakeholder. System Architect handles Model-Based System Architecting. Model-Based System Architecting requires Model-based Systems Engineering Environment. Model-Based System Architecting yields System Architecture.

1.1.2. A Tango of Conceptualizations and Decisions

A concept is an initial mapping of what we want to accomplish to the form that will be used to accomplish it. For example, "the rocket will land upright using stabilizer fins", "the vehicle will work on both fuel and electrical power", or "all the communications will go through the central message hub". The concept is part of the system architecture and should be specified appropriately within the scope of the MBSA process.

A concept maps function to form [13]. The function of a system is a process (an activity), which typically affects one or more operands (the objects that are changed by the activity). The form is a set of elements that support this function. This is analogous to the three core parts of all languages being the noun (instrument of the action), verb (activity that describes the action), and noun (the object of the action) [17]. Figure 3 illustrates the basic pattern of a concept and the association among the concept, function, form, and architecture.

The highest-level concept of the entire architecture should be a short phrase or sentence. For example, "Self Driving Car handles Transporting of up to 4 Passengers to a distance of 500km". In this short example we clearly see a) the process: Transporting, b) the form: Self Driving Car, c) the operand: Passenger (up to 4). Additionally, this statement includes an optional attribute: Distance (up to 500km), which may be drawn from some need.

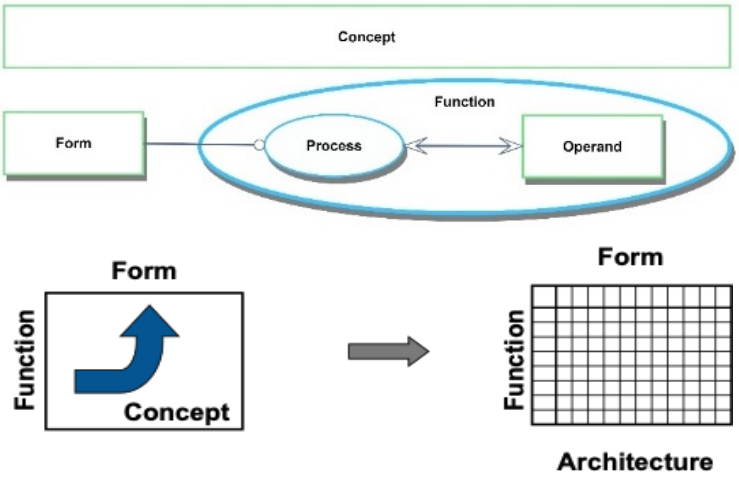
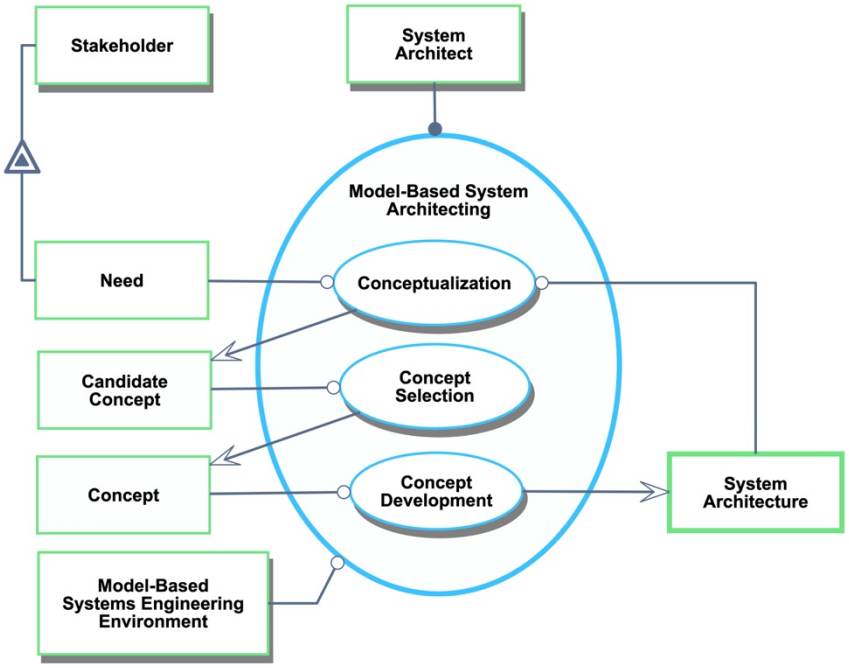


Figure 3. **Concept:** Form enables Function; Function = Process that affects an Operand. The allocation of Function to Form leads to an Architecture [13].

The long journey from needs to solutions passes through a series of steps and is by no means a straight line. Many of these steps go back and forth in what could be imagined as a tango dance. Many mental models have been proposed for this series of steps, most notably the V model and other examples [18], which mostly advocate a procedure of activities. We present a generic classification. We argue that at each point, architecting is either one of two cognitive tasks: conceptualizing or deciding. Conceptualizing is describing or specifying concepts, while deciding is selecting concepts from the available candidate pool. After deciding, the decision becomes part of the solution. Conceptualizing and deciding are collectively referred to as *reasoning*. Each architecting step is a reasoning step, and MBSA is a series of reasoning steps, as shown in Figure 4. Both types of reasoning – conceptualization and deciding – can benefit from a model-based approach.



1. Model-Based System Architecting zooms into Conceptualization, Concept Selection, and Concept Development, which occur in that time sequence.
2. Stakeholder exhibits Need.
3. Conceptualization requires Need of Stakeholder and System Architecture.
4. Conceptualization yields Candidate Concept.
5. Concept Selection requires Candidate Concept.
6. Concept Selection yields Concept.
7. Concept Development requires Concept.
8. Concept Development yields System Architecture.

Figure 4. Model-Based System Architecting in-zoomed

1.2. Model-Based Concept Representation

In this section, we discuss a conceptualization process, built around a concept representation framework, supported by OPM modeling language. In sub-section 1.2.1 we define an ontology for system architecting with five domains. Sub-sections 1.2.2 to 1.2.6 sequentially reveal each one of the ontological domains through a conceptual reference model. The iterative nature of the system design process is demonstrated through the interplay between the domains. The Scope of an MBSA Application is explained in subsection 1.2.7. We then use this framework to examine how MBSA changes the system architecting process.

1.2.1. System Architecture Framework

An ontology is a formal vocabulary of domain concepts. It is critical to adopt and formalize an ontology for a coherent discussion about concept representation, particularly for system architectures. Our ontology is illustrated in Figure 5. This ontology underpins a framework that introduces the core entries within the system architecture concept representation, and proposes ways to encode these entries, preferably in a modeling environment. The ontology consists of five domains: Stakeholders (D1), Solution-Neutral Environment (D2), Solution-Specific Environment (D3), Integrated Concept (D4), and Concept of Operations (D5). A concept domain is a subset of the ontology, which focuses on a specific aspect of the architecture, and has a mapping to other. Domains are distinguished by color. We list 28 entries within these domains, based on a concept representation framework introduced in [19, 20]. We reference these 28 entries using {EXX}, such as {E15} referring to Specific Form.

The first three domains, D1-D3, represent the simplest formulation of a concept. The fourth and fifth domains lie downstream to reflect a latent termination, i.e., extending D1-D3 as long as it is still appropriate to continue detailing the architecture. The exact timing for terminating varies with solution types and contexts [19, 20]. D1-D3 are distinguished from D4-D5 in the abstract vs specific levels of discussion. The system architect should be comfortable with switching from the abstract discussion (D1, D2, and D3) to a more concrete level of detail (covered by D4 and D5). Moreover, iterative system architecting means that D5 can impact D1, in a cycle of revising, diverging, and converging. Figure 6 shows the system architecture as a composition of the domains.

Domain 1 (D1) Stakeholders	
1	Stakeholder
2	Need

Domain 2 (D2) Solution-Neutral Environment		Domain 3 (D3) Solution-Specific Environment		Domain 4 (D4) Integrated Concept	
3	Solution-neutral operand (SNO)	8	Solution-specific operand (SSO)	17	Internal Operands (IO)
4	SNO value attribute	9	SSO value attribute	18	IO value attribute
5	SNO other attribute	10	SSO other attribute	19	IO other attribute
6	Solution-neutral process (SNP)	11	Solution-specific process (SSP)	20	Internal Processes (IP)
7	SNP attribute	12	SSP attribute	21	IP attribute
		13	Generic Form	22	Internal Elements of Form (IEoF)
		14	Generic Form attribute	23	IEoF attribute
		15	Specific Form	24	Structure
		16	Specific Form attribute	25	Interactions

Domain 5 (D5) Concept of Operations	
26	Concept of Operations
27	Operator
28	Context

Figure 5. A System Architecture Ontology and a Concept Representation Framework, adapted from [19, 20].

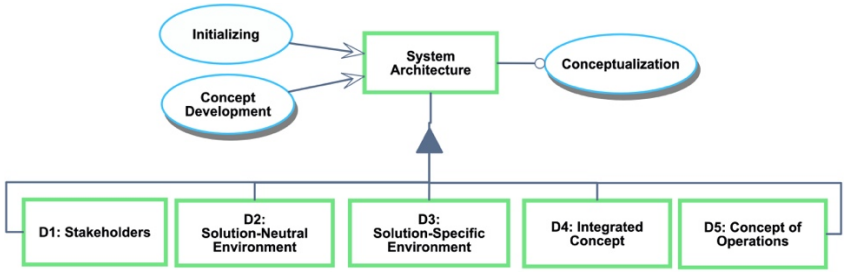


Figure 6. A System Architecture ontology with five concept domains.

1.2.2. The Stakeholder Domain (D1)

The Stakeholder domain (D1) captures stakeholders and their needs. A stakeholder {E01} is "any group or individual who can affect or is affected by the achievement of the system's objectives" [21]. In other words, many groups and individuals can be stakeholders in the broadest sense, depending on the context. This emphasized the importance of a broad operational context in which the system is intended to operate. Consider the European "Green Deal" [22]: according to the given definition, all humans on Earth are Green Deal stakeholders.

Stakeholder needs {E02} are defined as answers to the question 'What problems are we trying to solve?'" [23]. Needs should be problem-oriented and not solution-oriented. Needs are often specified before the system architect gets involved. Needs are often fuzzy, ambiguous statement by stakeholders. This fuzziness challenges system architects to clearly formulate the essence of the need, e.g., what is the expected capability, or expected outcome, or expected change to the current state. The special importance of the stakeholder needs is that they are used to formulate functional requirements in a solution-neutral environment.

The stakeholder needs might come from the variety of the sources. The first of them is the stakeholders themselves: this is the task of the system architect to frame the discussion with stakeholders in such a way that would help formulating those needs. Another potential sources of needs are the Use Cases, constraints, requirements that might come from extensive literature review.

The system architect's goal is to formulate the functional intent in each stakeholder need. Needs are associated with the problem statement first, expressed in the solution-neutral environment and realized through the process. Figure 7 illustrates the Stakeholders domain (D1) in which the stakeholders are denoted by rectangle and need is defined as an attribute (denoted by a black triangle inside a triangle) of stakeholders.

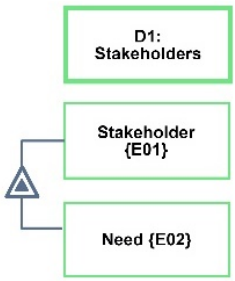


Figure 7. The Stakeholders Domain (D1): Stakeholder {E01} exhibits Need {E02}.

1.2.3. The Solution-Neutral Environment (D2)

The need for a solution-neutral environment is a fundamental design principle [24]. The solution-neutral environment (D2) facilitates the elicitation of functional requirements, which must be free of any bias towards prospective solution approaches, specific technical disciplines, or implementation strategies [25]. Therefore, the system architect specifies the essential information about the solution-neutral process before solution concept

development. The functional intent (such as "transporting passengers", "transferring money", or "playing music") should be formulated before the possible alternative solutions are set up.

Figure 8 encodes the solution-neutral environment (SNE) and its entries. The solution-neutral operand (SNO) {E03} is an object of interest that will undergo some transformation by the solution-neutral process (SNP) {E06}. The solution-neutral process manifests the dynamic nature of the function: it reflects the action. The SNP and SNO should be abstract so that a variety of alternatives will emerge and an informed decision-making process will take place. The SNE entries may have attributes, which are appropriate to start elaborating at this stage.

SNP {E06} maps to the need {E02}, which is specified in D1, as shown in Figure 8. Need is realized via the performance of some process – the SNP and the consumption, transformation, or generation of some operand – the SNO. Changes in need are likely to entail changes in SNP.

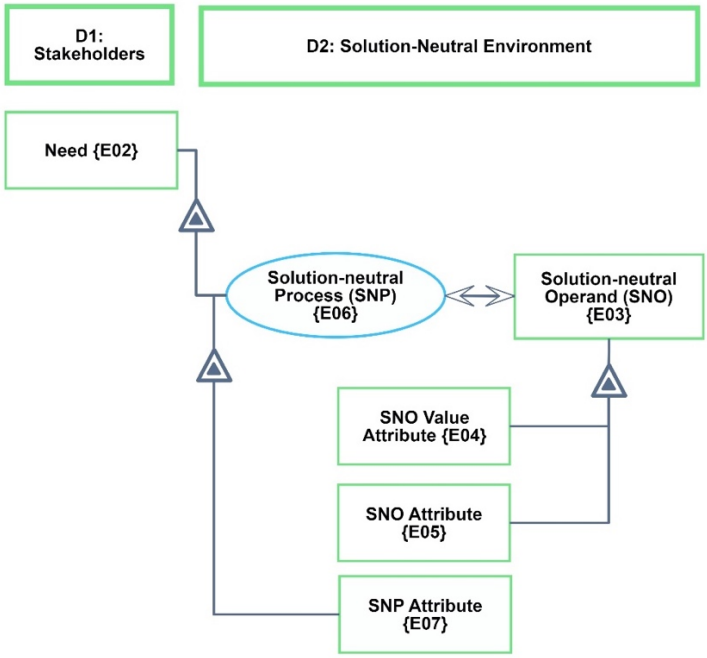


Figure 8. The Solution-Neutral Environment (D2): Need {E02} exhibits Solution-Neutral Process (SNP) {E06}. Solution-Neutral Process (SNP) {E06} affects Solution-Neutral Operand (SNO) {E03}. Solution-Neutral Operand (SNO) {E03} exhibits SNO Value Attribute {E04} and SNO Attribute {E05}. Solution-Neutral Process (SNP) {E06} exhibits SNP Attribute {E07}.

1.2.4. The Solution-Specific Environment (D3)

The solution-specific environment (D3) encodes alternative architectures. A solution entry defines how the system is going to perform the solution-neutral functions. Solution-specific processes (SSPs) and solution-specific operands (SSOs) are mapped to their solution-neutral counterparts (which were presented in D2), such that it is clear which solution-specific entry attempts to realize each solution-neutral one. For example, "Transporting by Air (Flying)", "Transporting by Land (Rolling)", and "Transporting by Sea (Sailing)" are SSPs refining the SNP "Transporting".

The solution-specific environment is derived from the solution-neutral one via the generalization-specialization relation (drawn as a blank triangle in OPM, as shown in Figure 9). The SNO "person" generalizes the SSO "passenger" in a transportation context, "patient" in a medical context, and "user" in a technological context. Domain jargon can better describe the artifacts, entities, and human roles (e.g., the SSO "exoplanet" in deep space exploration).

The number of possible solutions is a product of the number of D3 entries per D2 entries, therefore it increases with every additional solution-specific entry. However, the specification of solutions also narrows down the funnel of possible solutions. While the solution-neutral environment leaves room open for as many solutions as possible, solution-specific entries identify specific ways that realize the solution-neutral intent to choose from, and close the door to other unlisted ideas.

The solution-specific environment may also be associated with the principal solution – the deliverable of conceptual design [26]. A principal solution is a concept, and the early outline of an architecture. The key purpose of the solution-specific environment is to discover the architecture by specifying those forms as principal solutions. This is achieved by specifying Generic Form entities {E13} and associating them with the SSOs they enable or support. Every SSP has several optional Generic Forms that may implement it. This is a fundamental conceptual design principle, which, to some extent, further extends the solution space. For example, the SSP "Flying" can be implemented by several Generic Forms, e.g., Airplane, Helicopter, and Drone.

Each Generic Form can be specialized into Specific Forms (SFs) {E15} within the scope of the Generic Form. For example, "Jet Airplane", "Turbo-Prop Airplane", and "Propeller Airplane" are three SFs of the Generic Form "Airplane". The Vertical Take-off and Landing (VTOL) Aircraft concept, which is featured by Lockheed Martin's V-22 "Osprey", is a form with

lineage to two Generic Forms: Airplane and Helicopter. Therefore, a Specific Form which is a combination of several Generic Form can be a valid concept. In fact, converging multiple dimensions of Generic Form into a minimal set of specific forms is desired, if it reduces the tradespace into a smaller set of comprehensive, integrated solutions.

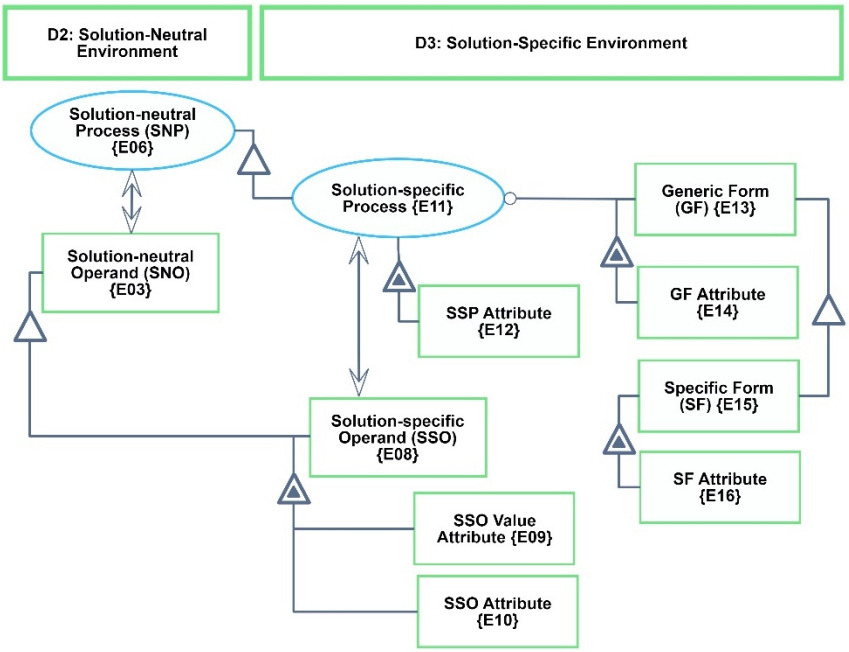


Figure 9. Solution-Specific Environment (D3): Solution-Specific Process (SSP) {E11} affects Solution-Specific Operand (SSO) {E08}. Generic Form (GF) {E13} enables Solution-Specific Process (SSP) {E11}. Solution-Specific Operand (SSO) {E08} exhibits SSO Value Attribute {E09} and SSO Attribute {E10}. Solution-Specific Process (SSP) {E11} exhibits SSP Attribute {E12}. Generic Form (GF) {E13} exhibits GF Attribute {E14}. Specific Form (SF) {E15} exhibits SF Attribute {E16}.

1.2.5. The Integrated Concept (D4)

An integrated concept fuses multiple concepts into a cohesive architecture. Two integrated concepts should be distinguishable from each other at a relatively high-level of abstraction (i.e., following a relatively small number of abstraction steps, such as the listing of internal processes or the breakdown into components). The integrated concept must also reach a sufficient level of granularity that allows for the critical transition from system

architecture to subsystem design [27]. The integrated concept actually results from decomposition, rather the composition, i.e. by increasing the granularity of the architecture.

The Integrated Concept Domain (D4) encodes the internal processes, operands, structures, and relations, as illustrated in Figure 10. Digital thread flows from the specific form {E15} in D3. Each function is compounded from an Internal Process {E20} and an Internal Operand {E17}. Internal Element of Form {E22} enables the functions. The structure (physical interaction of elements of form) and interactions (functional relationship of elements of form) between the system concept's entities are demonstrated at the bottom of Figure 10.

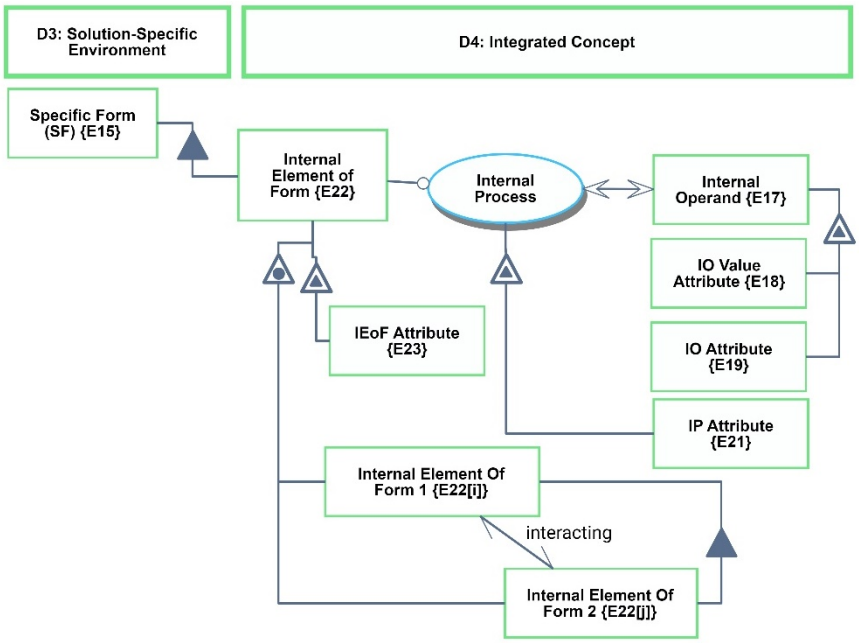


Figure 10. The Integrated Concept Domain (D4): Internal Process {E20} affects Internal Operand {E17}. Their attributes are specified {E21}, {E18}, {E19}, respectively. The internal elements of form {E22} is used to execute the function. IEoF's attribute is {E23}. The lower part specifies structural and interaction relations {E24} and interactions {E25} among instances of IEoFs {E22}.

A system architecture captures vertical and horizontal relations [28]. The vertical relations capture the decomposition or breakdown of systems into subsystems. The horizontal relations capture interactions between elements, such as flows of material, energy, or information. D4 caters to both the

vertical relations – encoded in the upper part of Figure 10, and the horizontal ones, encoded in the lower part of Figure 10.

The multiplicity of potential vertical breakdowns and horizontal interactions gives rise to the concern that the architecture of the integrated concept will become too complicated and messy. We should therefore set bounds on the amount of information to specify at this stage. Miller's Law states that an average human can hold 7±2 objects in short-term memory [29]. It has since become common to assert that 7±2, Miller's Magical Number, is a good limit for complexity, because constructs that include more than 7±2 items are likely to become difficult to grasp.

Completeness and complexity go together in our approach. That is to say: a complete integrated concept at the first level of decomposition (from a specific concept to the set of internal structures), is complete in the sense that it *utilizes its complexity quota*, so to speak: A view that comprises no more than 7±2 elements make a good candidate for completeness of specification. That is not to say that there cannot be more elements. More elements should be clustered with the existing 7±2 elements. Thus, 7±2 is in fact an estimate for sufficiency and a constructive measure of complexity, in the sense that it encourages the architect to converge on this range for complexity management. We can therefore say that a problem that does not converge on a 7±2 element scale at any given level of hierarchy, may not qualify for this approach.

1.2.6. The Concept of Operations (D5)

The Concept of Operations (ConOps) domain (D5) specifies the overall high-level idea of how the system will be used to meet stakeholder expectations [23]. The Department of Defense Architecture Framework (DoDAF) refers to the ConOps as a high-level abstraction graphic that captures how the system will operate, how it will work out together to help the operational stakeholders achieve their goals [30]. We have shown a similar model-based approach for analyzing the DoDAF Operational Viewpoint, which covers the ConOps [31]. The ConOps ties the system concept with the environment, and over time. The ConOps is important as it informs all stakeholders with the context and integrative operation of the system: what processes are to be performed, in which sequence, and how they will be executed by components of the architecture. Eventually, the purpose of the ConOps is to illustrate how the architecture delivers value. ConOps should include both the system of interest, and the accompanying systems that are necessary to consider during the system design process.

D5 focuses on the context represented through the whole product system {E28}, as shown in Figure 11. It includes the accompanying systems, system enterprise which is responsible for the system of interest {E15}, and operator {E27}. An operator is a person or group of people who operate the system. There is always one higher level in which an architecture resides, unless we aim to architect a universe, which, to the best of our knowledge, is the most inclusive architecture of all.

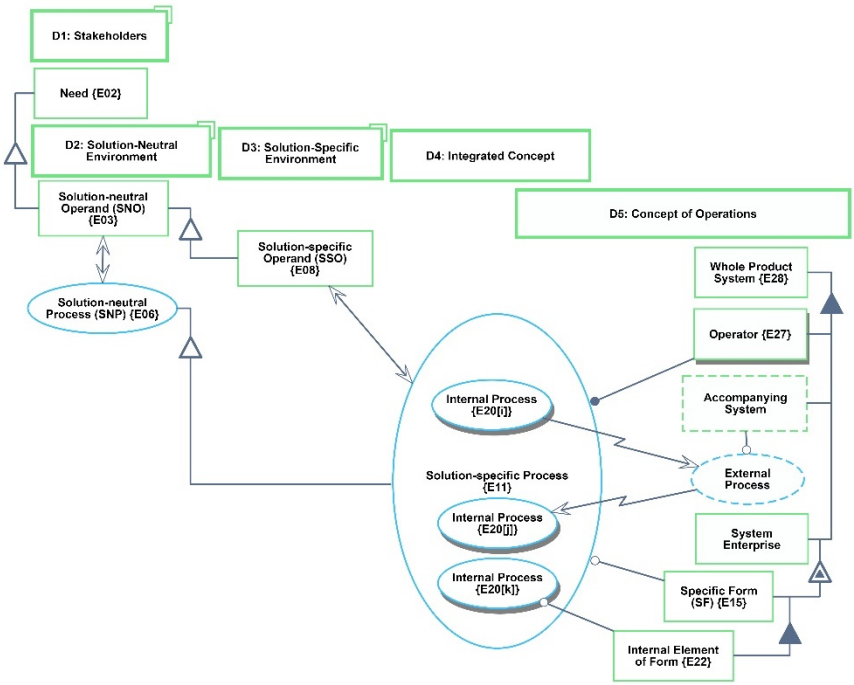


Figure 11. The Concept of Operations Domain (D5)

The context defines how the system interacts with its environment. The same architecture can perform perfectly in one context and poorly in another. For example, a Formula 1 racer will be amazing on the racing tarmac but less adept on the loose surfaces of the Dakar Rally. Even if the architecture remains the same, the context provides the boundaries and constraints in which the solution architecture must operate successfully.

While it might make sense to consider the ConOps earlier in the process, it can also be harmful because setting too many constraints and restrictions limits our ability to come up with good solutions. Consider, for example, that operational stakeholders will impose a ConOps that heavily relies on manual or cognitive actions, while the whole solution can be autonomous or

semi-autonomous. We would rather explore multiple options and validate autonomous solutions by finding an appropriate ConOps, rather than try to fit it into a human-intensive environment. Thanks to a paradigm shift in the automotive industry, many more functions are delegated to automation and relieve the driver of the cognitive load rather than intensify the burden.

It is still possible to specify ConOps upfront for an already well-defined operational architecture in which the architect has to integrate a new capability or functionality. The specification of legacy elements, platforms, and reusable assets can be helpful both for solution-specific concept viability and for further-up elicitation of needs. In some cases, a critical analysis and challenging of existing operational concepts may help elicit the true underlying needs of operational stakeholders, which may open up the door to other major architectural enhancements.

ConOps has connections with the other domains, which is illustrated in Figure 11. The iterative nature of the system design process is embodied in the clear digital thread that starts with stakeholders and their needs and culminates in D5. Figure 11 demonstrates the role of D5 in context representation, as well as inclusion of the system design process in a coherent way in which the domains are interwoven to deliver a value from system operation to meet stakeholders needs.

1.2.7. The Scope of an MBSA Application

The scope of an MBSA project may be a subset of our framework. The system architect may choose to focus only on some domains, depending on how broad or narrow an exploration they desire. Ideally, we would try to model just enough to have a reasonable evaluation of our architectural options, identify evaluation criteria, and move forward with an architecture. MBSA should capture sufficient detail to support the detailed design. Unfortunately, the broader the architectural decisions under consideration, the more general the models must be to account for the breadth of options. The presented framework assumes that the fixed effort available in the architecting phase is a tradeoff between breadth and depth of architectures evaluated.

MBSA is designed to minimize unnecessary effort. If, for instance, a solution-specific environment (D3) is already defined due to various constraints (for instance, implementing some functionality using specific hardware type), we may skip the divergence from solution-neutral environment (D2) and attempt to match the solution-specific environment (D3) with stakeholder needs (D1). Solution-neutral and solution-specific functionalities are defined in a way that clarifies and simplifies the MBSA effort. This

approach also helps systems architects focus on those functionalities that are most critical to constitute decision points that would direct the architecture one way or another.

System architecture, like civil architecture, is both science and art [32]. This scope should answer the following questions, including explicitly specifying what lies outside the boundary of MBSA:

1. Which functions must, should, should not, and must not be captured?
2. Could introspecting on the functions of interest yield a re-formulation of the problem? If so, how general must the model be?
3. Which components should and should not be captured?
4. How do we determine if someone is a stakeholder and whether they should be included?
5. How do we evaluate synergies or conflicts in a given architecture?
6. What are the insights derived from the process of architecting beside the outline of a selected architecture, and how do we preserve those insights in order to further inform the design process?
7. At which level of granularity is it sufficient to decompose the system of interest in relation to context and specific needs of stakeholders?

MBSA allows for recording the answers to these questions within the architecture model, and within the context of our concept representation framework – thus extending and empowering the cognitive process done by the system architect in order to consider and answer these questions. Indeed, just like a painting is an artifact of the artistic process, a model can record the emergent propositions that we include in a system architecture, such as elegance, empowerment, holism, and inspiration – all of which are subjective perceptions that we hope stakeholders will experience when presented with the selected system architecture.

1.3. MBSA And Architectural Decision-Making

In this section we focus on the value of a model-based process for architectural decision-making. Architectural decisions are those reasoning steps that affect the direction in which an architecture evolves. Decisions are made throughout the process, and some are based on the model. We focus on how model-based system architecting would be different from a traditional "offline" system architecting process. It has been generally asserted that models promote easier design reuse, evaluating more options, and automating design space exploration. We are interested in a deeper question of how we might expect decision-making to change. The mere availability of models has not broadly changed the decision-making process. We ask what is it

about MBSE capabilities that would lead us to believe that the process of choosing among potential architectures is different, and better?

R&D stakeholders often make incremental decisions on large programs, before detailed solution comparison tables are available. The front end of design is often ambiguous, and there is a gulf to be crossed between the concept-as-a-napkin-sketch and detailed specifications. We need more rigorous bridges than the traditional ones – slideshow illustrations and overly complex draft box-and-line drawings.

The ability to visualize decision-supporting information as discussed in the previous section had taken a back seat while the MBSE community had focused on the modeling environment and the user experience of the modeler or analyst. Product managers and engineers have been somewhat neglected and have lost their ability to look at a model, recognize a dilemma, understand the options, and make or at least advise a decision. In this section we try to remedy this situation by focusing on decisions rather than on excellence in modeling.

Following a brief discussion of some decision-theoretic concepts (subsections 1.3.1 and 1.3.2), we study several ways in which MBSE enhances architectural decision-making: Capturing stakeholder needs as cost and benefit manifestations of architectural decisions (sub-section 1.3.3); Capturing and discovering the tradespace of possible candidate conceptual architectures, and highlighting decision points and inviting the architect and stakeholder to resolve them; Specifying architectural decisions by detailing the solution-specific architecture in the context of the problem domain; and highlighting the decisions that were made or will have to be made throughout the architecting process, their impact on the evolving architecture, and the trace of justification and rationalization of the emerging architecture (sub-section 1.3.5).

We consider driver behavior tracking, an issue that vehicle owners are familiar with, as they want to ensure the safe and lawful behavior of those who drive their vehicles. This issue is well known to vehicle fleet operators, rental companies, insurers, and parents of adolescent children. We would like to find a solution for this problem.

1.3.1. What is a decision and which decisions are architectural?

Decisions are the choices that one makes about something after considering several possibilities [33]. This definition emphasizes that a) each

decision emerges from several alternatives, and b) the choice should be made after reasoning and consideration. A decision is the outcome of a decision-making process.

Architectural decisions are those important and critical-to-make decisions that have a significant impact on the *concept* – i.e., a significant transformation of system structure and behavior [13]. Our cognitive and mental abilities and subjective biases may make the most important decisions indiscernible from less important ones. The model-based approach helps place stakeholders on the same page and ensure that priorities, impacts, and implications are clear to all, conventionalized, and objective, as part of the decision-making process.

Decision-making is the process of reaching a decision. It generally consists of three phases: Decision Problem Definition, Deciding, and Decision Execution. A more detailed description of the canonical outline of decision-making is provided by [34].

The system architect's primary role is decision-making, and decision making is the essence of architecting, however, more and more architectural decisions are made in groups, and the architect's role becomes one of facilitating, moderating, informing, and recording architectural decisions [35]. This notion highlights the importance of a suitable platform that would assist the system architect throughout the architectural decision-making process. Decision support capabilities include information management, formulation, recommendation, selection, execution, and learning [36].

The relevance of several alternatives is natural to humans. From the most trivial chore to the most pressing and fundamental issues of our lives, there are always at least two options, and even when there is one visible option, there is also a shadow, or default option of "doing nothing" (DN). When we consider medical treatment, we identify alternative clinics, physicians, medical approaches, available days and hours, healthcare coverage, and the risk of worsening our medical condition. Complex system architecting is no different: When we design a new aircraft, we evaluate the desired capacity, fuel consumption, range, piloting automation capabilities, situational awareness, etc. Alternatives emerge from key attributes, relevant values, and feasible combinations.

Decision analysis is the scientific foundation of decision-making. It is rooted in both the exact and social sciences, giving rise to two DM paradigms: the analytical and the behavioral. Analytical, model-centered approaches emerged from classical probabilistic and utility-theoretic approaches and focused on rational choice [37–39]. Behavioral decision theories view decision-making as a non-normative, human-centered

process, with all the issues it raises [40]. According to the behavioral school, a person or group of people make decisions under various constraints, uncertainty, and bias [41]. The behavioral approach deals with heuristics, rationality and rationalization, analysis paralysis, and a host of other aspects and phenomena of human cognition. Managerial aspects like decision tracking and assurance are considered mainly behavioral.

Two primary handbooks on systems engineering (INCOSE's and NASA's [23, 42]) discuss decision making as an engineering process, concerning both programmatic and architectural aspects. Programmatic decisions are made at decision gates, to simplify project and risk management. Architectural decisions concern aspects like functionality, design, technology, and vendor selection.

Trade Study, or Tradespace Exploration, is the process of analysing various architectural alternatives, and trading-off figures of merit until a balanced solution is obtained [43, 44]. The primary phases of a trade study are: problem scoping, communicating with stakeholders, defining evaluation criteria and weights, defining and filtering alternatives, evaluating candidate alternatives based on measures of merit, selecting the best alternative, reviewing and re-evaluating the selection, assessing impact, and validating assumptions.

Reasoning about the alternatives is a fundamental part of our actions. There is a need to support the reasoning process through the recording and visualization of alternatives, compositions of alternatives, and comparisons of alternatives. The model-based approach empowers reasoning by reducing all the alternatives into a common formal language and representation. The representations of alternatives are not always straightforward, and the need to formulate them under the syntax and semantics of the modeling language makes the alternative concepts and their attributes emerge through an interactive cognitive-computational process, rather than through a deterministic mechanistic process of converting ideas to models.

Additional concerns related to decision making involve the consideration of uncertainty, risk, and subjective bias. Being able to integrate requirement prioritization, solution architecture selection, design exploration and decision making, and risk analysis is the essence of informed decision-analytic system architecting [14]. We would add here that the ability to do all of this in a model-driven way and in a model-based environment, particularly during early conceptual architecting iterations, is the essence of this chapter.

While the concept is a function-form mapping at the highest level of abstraction, a series of architectural decisions maps a set of functions to their respective forms. Consider an example of the functional intent of Money

Moving, as illustrated in Figure 12. In this case the stakeholder {E01} is defined as a *Person*, who has a need {E02} *Change the location of money from point A to point B*. The solution-neutral process *Moving* {E06} affects the solution-neutral operand *Money* {E03} by changing its state from *Point A* to *Point B*. The solution-neutral process *Moving* {E06} can be interpreted as either *Transporting* {E11.1} (moving money in the form of bills and coins in a physical way, e.g. using a secure courier), or *Transferring* {E11.2} (offsetting sums of money electronically between two accounts, sometimes commonly called *wiring*, after an ancient legacy dating back to the days of the telegraph). Both the physical *Transporting* and the digital *Transferring* are solution-specific processes that lead to completely different alternative architectures for delivering money to or for stakeholders: *Flying Vehicle* {E13.1.1} and *Land Vehicle* {E13.1.2} in case of *Transporting* {E11.1}, and *Wire Transfer* {E13.2.1} and *Check Deposit* {E13.2.2} in case of *Transferring* {E11.2}

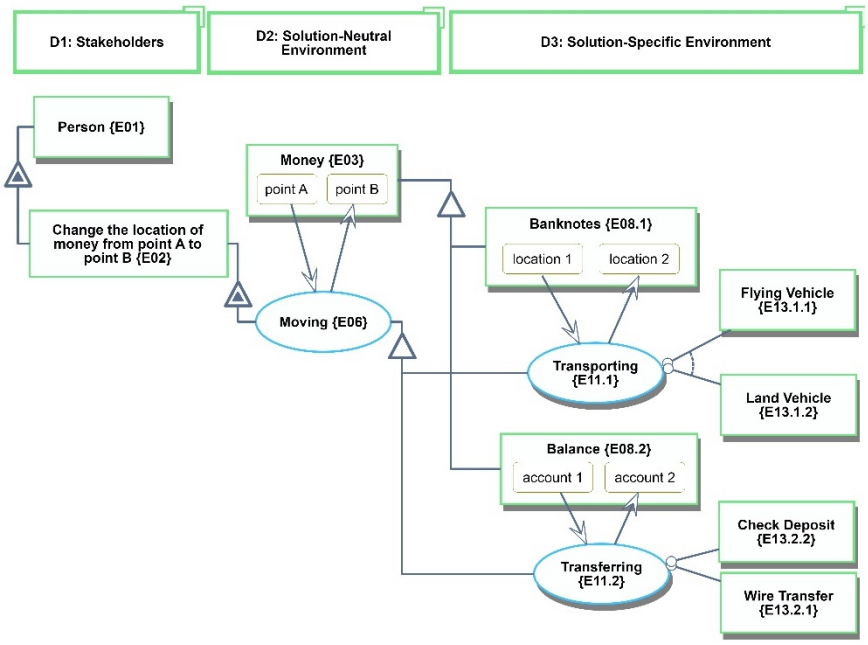


Figure 12. Architectural choices for the solution-neutral functional intent of Money Moving.

The distinction among SSPs is not always obvious. *Transferring* can also be interpreted in a physical sense, for example, through a *Check Deposit*. Further decomposition of SSPs can quickly reveal the distinctions between alternative concepts for realizing the original solution-neutral functional

intent. The Internal Processes of *Transporting* and *Transferring* reveal the differences between these two conceptually distinct solution-specific architectures. The physical steps are distinctly different from the digital steps needed to complete the Money Moving procedure under each architecture.

The common representation and formulation of the two distinct approaches using the same modeling framework enables reasoning about these approaches. Placing these abstract concepts together and showing how they are both derived from the same solution-neutral functional intent, facilitates the cognitive process of comparing alternatives. The system architect can go back and forth between the solution-neutral and solution-specific environments. The common representation facilitates discussion with stakeholders about the alternatives to validate the functional intent, to discover new alternatives, and to eliminate the irrelevant ones. This iterative process leads to continuous improvement and validation of the emerging architecture.

1.3.2. Concept Attributes, Metrics, and Decision-Supporting Criteria

We now discuss ways to include decision-supporting criteria for concept evaluation as part of the MBSA process. The terms criterion and metric may be used interchangeably, but while a metric is typically perceived as a general-purpose quantitative index (e.g. the Dow-Jones, the outside temperature, or the second moment of inertia), a criterion is defined in the context of a decision problem and is usually weighed against at least one other criterion. Both criteria and metrics can be qualitative or quantitative. Qualitative criteria must be ordinal or ranked, such that it is clear which value is better. That said, two stakeholders could aspire for opposite trends of the same criterion. For example, airports want to maximize the volume of air traffic while nearby residents want to minimize it. A binary criterion is either high or low, met or unmet, true or false, success or failure. A ternary criterion has three levels, e.g. high-medium-low, red-orange-green (also known as a traffic light criterion, etc.), and so on. Any number of ranks can be applied to a criterion. However, higher separation of ranks, require more precision in the induction of the value such that one can be confident about the suitability of the ranking assigned to an alternative in a specific criterion.

Criteria depend on the context and vary from concept to concept, and across stakeholders. Nevertheless, all criteria can be encoded in the model-based framework and support decision-making about the system architecture throughout the process. We can capture any criterion as an attribute of solution-neutral environment (D2), solution-specific environment (D3), and

integrated concept (D4). A value-related attribute is the one which is changed by the associated process.

Consider, for instance, a dilemma between three vehicle architectures: front-wheel drive, rear-wheel drive, and all-wheel drive. The question is not merely about mechanical feasibility – all options are feasible, and all have uses and applications, and therefore the evaluation must be based on the needs. Recognizing this situation as a decision problem is the first stage in the decision-making process. The decision problem should account for metrics that are sufficiently detailed on the one hand but sufficiently design-agnostic on the other hand. This is perhaps the essence of the distinction between architecting and designing. The decision may rely on cost, size, weight, weight distribution, volume, power consumption, maintainability, etc. – as long as we can assign values to such criteria through elaboration of candidate solution architectures. The concept representation framework encodes the key performance metrics (which serve as decision-supporting criteria) as the attributes of the solution-neutral domain (D2), solution-specific domain (D3), and integrated concept (D4). Specifying the attributes is therefore critical for alternative evaluation and comparison and not only as additional information. The MBSA framework enables of any system concept analysis that the system architect and design team may come up with.

1.3.3. Capturing stakeholder needs

Any spreadsheet would do for listing stakeholders and needs, but a model-based approach enhances this process in several meaningful ways. Model pattern reusability is useful for identifying stakeholders. A pattern of stakeholder types could include broad stakeholder categories and roles (owner, operator, regulator, etc.), as shown in Figure 13.

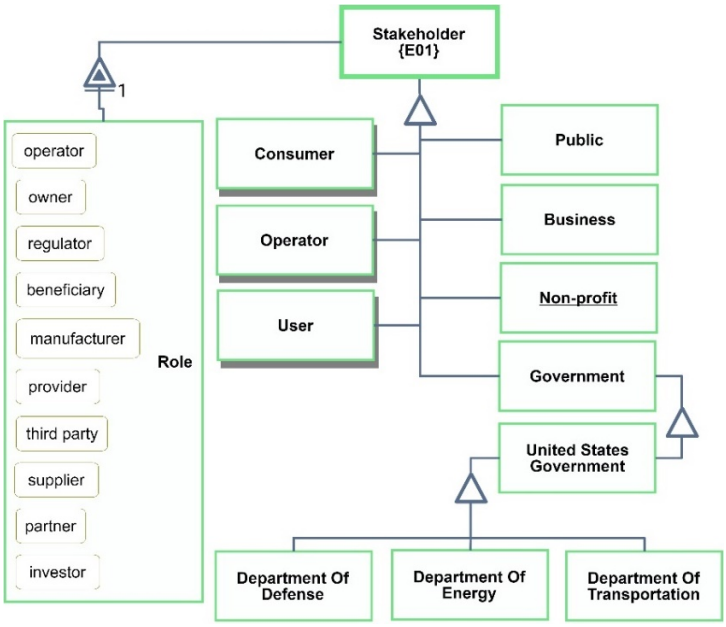


Figure 13. A reusable pattern of stakeholders

A dependable and reusable pattern of stakeholders – a decomposition and specification of subdivisions and attributes of the stakeholder concept – can be very useful in quickly and efficiently choosing stakeholders, rather than rediscovering and recreating them for each model anew. A stakeholder pattern may include both high-level and lower-level roles, units, and profiles, as well as generic or umbrella needs. For instance, a national or regional energy authority obviously wants to provide energy to its residents, while minimizing operating costs and obeying the law. They might also care about clean and sustainable energy generation solutions. Such needs can be encoded in a pattern and utilized for various applications, even such that only consider the energy authorities indirectly, e.g., mass transportation projects.

Stakeholder entries may also be elaborated with domain terminology. Ffor example, a defense stakeholder pattern can draw from the DoD Architecture Framework (DoDAF) [30]. Other aspects could include geographic distribution, available resources, standards, regulations, laws to comply with, strengths, weaknesses, opportunities and threats (SWOT), and so on.

Specifying stakeholders in the model, whether through discovery, documentation, or reuse, informs the system architect about all those parties and people who might need to weigh in on problem and solution domain

decisions. Stakeholders who are not the primary customers or beneficiaries of an architecture must also be identified, and their needs must also be well-understood, especially if they are in tension. A stakeholder set also leads to an emerging stakeholder network. Mapping stakeholder relations can be done in stakeholder value networks [45].

Consider for instance the Israeli missile defense system "Iron Dome", which has the intent of protecting the country against incoming rockets and ballistic missiles. Thanks to the protective umbrella that this system has cast over rural southern Israel, the economy in those areas began to thrive. Industry, commerce, and tourism indirectly became stakeholders. Nationwide franchises have had a chance to lobby on the deployment of Iron Dome batteries, due to the national economic impact these have had on their business. Before the system had been field-tested, it was very difficult to find deployment locations for launchers, sensors, logistic support, command and control outposts, and military encampments to support the massive operation of Iron Dome. Years later, shopping centers with a piece of the system on their outskirts suddenly became attractions. Having all stakeholders and needs in a common model could have led to different dynamics.

Institutional stakeholders often elaborate their needs as so-called stakeholder requirements. We include those as identified stakeholder needs, which simplifies the process, however there is a caveat. Stakeholder requirements are not really requirements. While this assertion may seem rude to some readers who have been stakeholders, this assertion stems from the understanding that both parties – stakeholder and system architect – are interested in getting to the stakeholder's essential needs, prior to establishing any solution that would meet those needs.

A model supports a hierarchy of interrelated needs that serve to justify those bottom-line or most central needs that stakeholders chose to state as their expectations from the system. For example, the Vehicle Owner in our Driver Behavior Tracking example ultimately wishes to maximize vehicle utilization while minimizing the risk to the driver and vehicle. Business owners might also be interested in monitoring schedule compliance by their professional drivers. We can figure out together with the owner or, say, the consumer association as a representative organization, how technology can help, next to other approaches like education or regulation. An in-vehicle technology could focus on collecting, analyzing, reporting, or acting upon drive and driver behavior characteristic data.

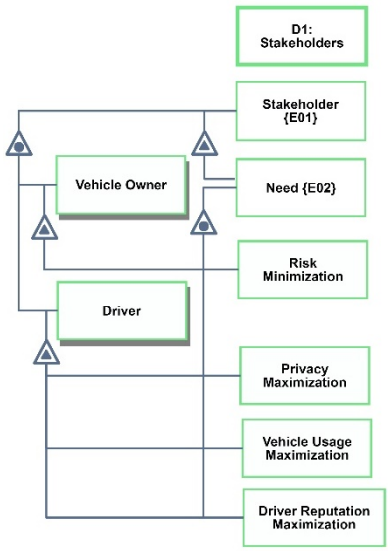
The presence of requirements in a model is truly informative when they serve as references for architectural decisions – regardless of how needs and requirements are captured. Systems engineers are responsible for

traceability – showing that each requirement (or need) is mapped to pieces of the solution, so that stakeholders (and especially customers) may track and verify the fulfilling of their expectations from the system.

Decision-making is a process of choosing one possible solution out of a population or solution space, according to some criteria by which the solutions are assessed, scored, or ranked. Stakeholder needs constitute such solution assessment criteria because they capture the benefits and costs that stakeholders will reap from any given solution.

In driver behavior tracking, the owner is obviously a stakeholder, but also the driver, the insurer, the vehicle manufacturer, and the regulator. We might say that the public is also a stakeholder, in case the owner is a public entity such as a consumer-serving delivery provider, government agency, law enforcement agency, non-profit organization, etc. For simplicity, we will focus on two stakeholders: Vehicle Owner and Driver. Figure 14 illustrates the instantiation of D1 with our two Stakeholders and their Needs.

The Vehicle Owner needs to minimize the risk to the vehicle, driver, and passengers while the driver wants to maximize her use of the vehicle and maintain a good reputation as a responsible driver. The driver is a central stakeholder in our case. One of those needs, for example, is Privacy. In some settings, drivers' right to privacy exceeds the vehicle owner's right to monitor their vehicle. For instance, car rental companies must not spy on their customers (in most countries), and employers may not be allowed to monitor their employees beyond scheduled work hours. We may not invade the right to privacy, and therefore some practices, e.g., in-vehicle ambient voice recording, may not be permitted. On the other hand, for law enforcement and public safety agencies, operational activity monitoring and debriefing is a critical activity which may significantly benefit from such a capability. It is therefore clear that while such a privacy-violating capability may be useful for driver behavior monitoring, it must account for the circumstances and might not always be applicable. These notions impact our architectural decision-making process by illuminating the tradespace about such aspects, thereby validating the architectural decisions to follow.



Driver and **Vehicle Owner** are instances of **Stakeholder, E01.**

Driver Reputation Maximization, Privacy Maximization, Risk Minimization, and **Vehicle Usage Maximization** are instances of **Need, E02.**

Vehicle Owner exhibits **Risk Minimization.**

Driver exhibits **Driver Reputation Maximization, Privacy Maximization,** and **Vehicle Usage Maximization.**

Figure 14. Stakeholders of the Driver Behavior Tracking System are affected by the main functionality of the system: Driver Behavior Tracking.

We can articulate needs as global metrics with aspired trends, e.g., Risk Minimization or Privacy Maximization. This approach prepares the needs for multi-attribute utility analysis [46] and multi-attribute tradespace exploration (MATE) [47], which aggregates all benefits and costs. This approach also helps in discovering opposite and biased needs and objectives. For example, in the case of privacy, the owner may not directly wish to violate the driver's privacy, but their need to receive as much information as possible about driver behavior may eventually compromise driver privacy. It does not mean that the owner has a need to "Minimize Privacy". If stakeholders had two conflicting needs, i.e. opposite trends on the same metric, we can quickly see how these might give rise to a potential conflict. For example, assuming that driver behavior tracking is not a passive process from the driver's perspective, the driver might want to "Minimize Data Collecting" while the vehicle owner might want to "Maximize Data Collecting" in order to gain as much information as possible. One way of resolving such conflicts is by weighing the alternatives and understand the fundamental needs. In our case the driver's fundamental need is actually to minimize disturbances. Such stakeholder needs revisions may eliminate inherent bias in need specifications, and resolve conflicts.

The specification of solution-neutral functions is mapped to needs and solution-neutral functions (SNPs), as shown in Figure 15. Once clearly defined, we should have a problem-domain, solution-neutral model, covering the stakeholders, their needs, operational activities, and operands of interest.

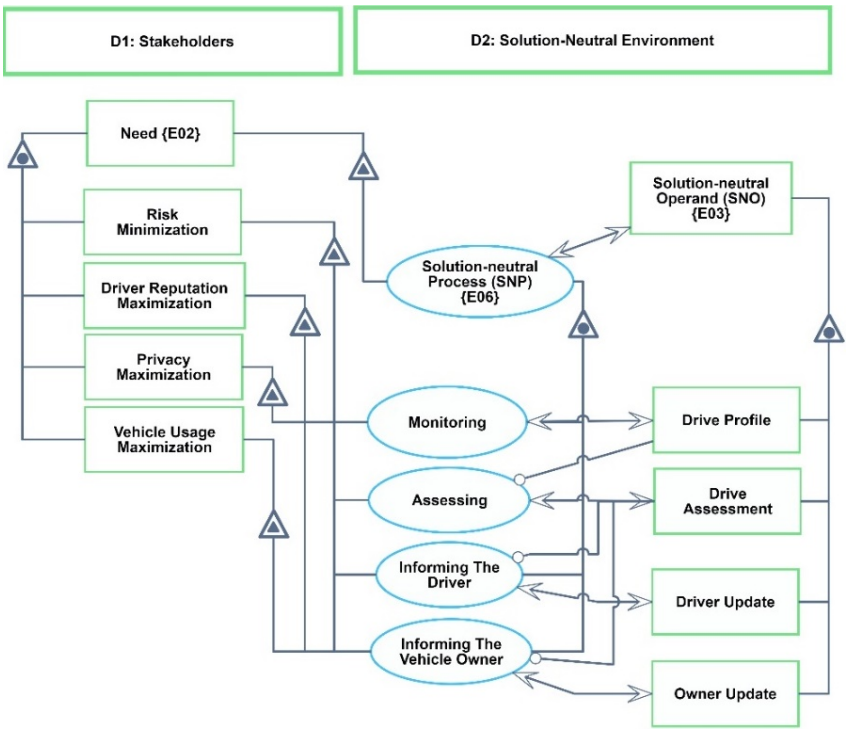


Figure 15. Mapping stakeholder needs to solution-neutral processes, and solution-neutral processes to solution-neutral operands.

Presenting the problem model to stakeholders allows them to validate a formal concept representation and clarify their needs in a common language. This approach is better than any approach in which there is no shared model that ensures consistency, common language, and holism. A stakeholder-driven model of the problem-domain is a necessary but not a sufficient condition for supporting a solution space generation process. In fact, failure to correctly represent the problem domain with a model may result in converging on a limited set of feasible solutions, which raises the risk of an invalid solution. The narrower the problem-domain description is, the likelier the solution architecture to be a bad solution for the wrong problem.

1.3.4. Capturing and discovering possible architectures

The system architecting process can be structured as a series of decisions [13]. Key to this idea is identifying and filtering forward those decisions that have the greatest impact on the system's performance and cost. We would like the model to assist us in confining the scope to the necessary minimum so that we would be able to focus on the critical decisions and the supporting conceptualization to inform them.

The number of potential architectures under consideration is theoretically the product of the numbers of options per decision. For example, if we have two decision variables and each decision variable has two options, then the total number of architectures is 2x2=4. If a third decision emerges, with, say 3 options, the number of integrated alternatives becomes 2x2x3=12, and so on. Many combinations may be logically infeasible and therefore excluded, but we can still end up with a large number of combinations. Figure 16 illustrates the mapping of architectural options to architecture decisions, with three decision points (A, B, and C), each with two options. Theoretically, we have a total of 2x2x2 = 8 options. We also illustrate three possible system architectures: SA1, SA2, and SA3. We map each architecture to the options that it relies on per each architectural decision.

Table 1 summarizes all possible combinations of decisions to be made. Three out of the eight combinations have been identified as relevant candidates for further exploration. Generating the combination table from the model is an important capability for ensuring consistency between the map of the problem domain and the list of applicable integrated solutions. It is also critical for ensuring that suitable, reasonable, and feasible alternatives are considered and not only the obvious, immediate, or convenient ones. Consider, for instance, that combination 6, that we can encode as a vector [a2 b1 c2] vis a vis the vector of options [A B C], is a truly brilliant combination that has not even been considered. Furthermore, if a new, previously unthought-of, or neglected decision variable is added, it brings the number of combinations to 16. This could be a game changer in many ways.

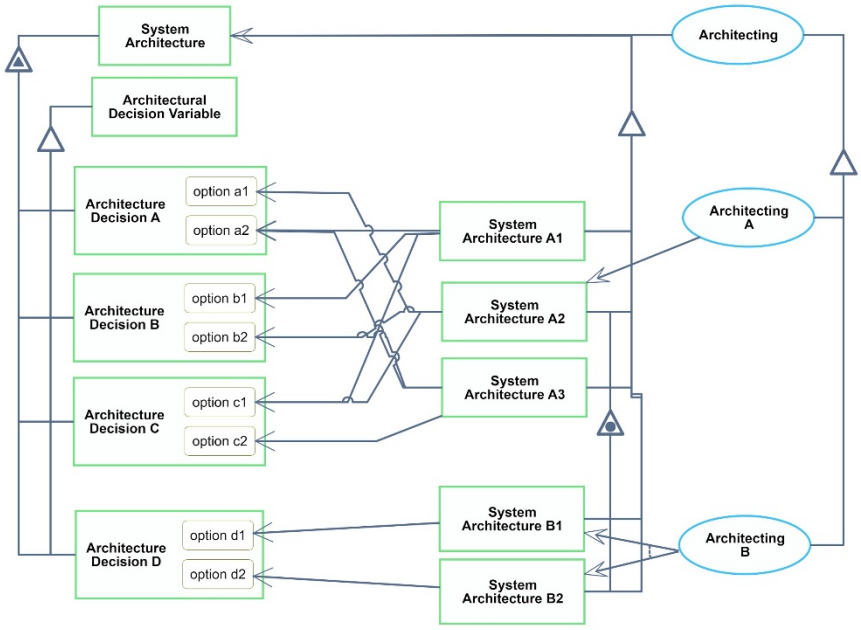


Figure 16. Combinations of architectural decisions generate as many architectural candidates as the product of the numbers of options per decision variable.

Table 1. Three out of eight potential combined architectures based on three decisions with two options each.

Combination	Arch. Decision A	Arch. Decision B	Arch. Decision C
1	a1	b1	c1
2	a1	b1	c2
3 - Architecture 2	a1	b2	c1
4	a1	b2	c2
5 - Architecture 1	a2	b1	c1
6	a2	b1	c2
7	a2	b2	c1
8 - Architecture 3	a2	b2	c2

Maintaining a valid table of options, based on an evolving model, even after an initial conceptual architecture is determined, has tremendous importance in ensuring the validity, sufficiency, and completeness of the tradespace, and consequently the validity of the tradespace exploration

effort - both by ensuring a consideration of all feasible options and by continued validation of the tradespace.

A retrospective of the Apollo Program considered nine possible decisions (including Lunar-orbit Rendezvous, fuel type, and other considerations) [13]. If each decision has only two options, then the number of possible combinations is 2^9, which is 512 architectures. Each option must be enumerated and considered, even if only superficially, in order to filter out irrelevant options and converge on a small and manageable set of combined options. We could map five additional candidate architectures to all remaining possible combinations of decision options, but with 512 alternatives, this seems to be impractical.

Enumerating the alternatives is only the first stage. We must determine each candidate solution's feasibility by some study and elaboration of details that will allow us to reach a confident conclusion about each candidate architecture in order to decide if we keep or drop it.

The set of decisions can also be used in generating placeholders for all options and then selecting candidates to populate. We can filter out options which are unlikely, infeasible, or too expensive. We can consider the three architectures illustrated above as the three finalists that survived the filtering process out of the original eight (possibly following thorough analysis using our concept representation framework). We will return to these options later as we elaborate them from the placeholder level to full-blown architecture specifications, relying on structural and functional building blocks.

There are situations in which options are implicit in the choice to regard or disregard an artifact. Consider, for instance, a specific stakeholder's requirement about compliance with some protocol, which is not mission-critical, but potentially a good idea in terms of interoperability, reusability, and risk reduction. We can easily relax the assumption that protocol compliance must be obtained by specifying two optional states for that protocol as either accepted or deferred/rejected. This step will immediately inflate the tradespace by a factor of 2 because each available combination will have to be assessed with and without the protocol. Thus, we can significantly expand the tradespace by referring to any predetermined concept as possibly-redundant. The binary enumeration of option states is therefore critical to comprehensive coverage and enumeration of tradespace options.

We can reduce the tradespace by disqualifying either the accepted or rejected option for binary decisions, or by splitting the decision process into a series of decision points, in which each decision point consists of a subset of the decision variables, resulting in more decision steps but significantly

less options to enumerate, assess, and choose from. Listing the options in the model, evaluating a subset of combinations, and selecting those combinations we wish to explore further, makes this process significantly easier, smarter, and more consistent.

The impact of MBSA here is therefore in the ability to enumerate candidate alternatives based on a model of architectural decisions. By specifying options as states of the artifact, we facilitate a process of enumerating the candidate alternatives.

For driver behavior tracking, we might consider, for instance the following aspects as fundamental to determining a conceptual architecture:

1. One size fits all vs usage-specific variants - this would imply a single product versus a product line - and will significantly affect the architecture of the product platform and the variants.
2. Open vs closed sensor policy - are we going to allow a variety of sensors to plug into our solution or only one or two specific sensors.
3. Vehicle-integrated vs stand-alone solution - are we going to embed the system in the vehicle and fully integrate it with vehicle systems such as the dashboard displays and vehicle component bus, or install it separately (possibly with a small connection to the vehicle for basic monitoring or interfacing capability)
4. Driver management vs anonymity - are we going to include driver identification and personalization, such that data and behavior patterns are directly associated with a specific driver, or leave the task of figuring out who drove the vehicle while misbehaving without the assistance of our technology.
5. Driver notification available or not - are we going to alert the driver, or only collect the data and report to the subscribing customer (who could be the owner, the driver, or the insurer but it would not be a real-time indication).

We could continue defining more aspects of an architecture and gradually increase the tradespace. We already have 32 combinations if each decision variable only has two options. We can also refer to these issues as five serial decisions and only consider one variable at a time. This will result in evaluating 10 solutions in total - 2 per step. The risk is in missing potentially preferable solutions hiding among the other 22, by discarding possible combinations by nailing down one variable after another. With 6 variable serial

decision-making, the ratio is 64 combinations to 12 inspected solutions and the difference is 52 ignored solution candidates.

For N binary decision variables the ratio between exhaustive search and serial decision-making is $2^N : 2N$. While serial decision-making seems inherently sub-optimal, this is how many of us make architectural decisions - resolving one issue or a couple of issues at a time. A model-based approach allows for both visualizing and analyzing the problem space with clear understanding of the implications of breaking down the problem into a series of smaller problems, as opposed to thoroughly studying the entire state space. Reaching a compromise is often a good idea, but it still requires good understanding of how decision variables can be grouped together into conceivable subspaces of the entire tradespace.

1.3.5. Capturing the architectural decision-making process alongside the resulting architecture

The analogy between systems architecting and multi-criteria decision-making decision-making can be formalized using Category Theory. Category Theory is a branch of mathematics that focuses on the equivalence of representations and transformations of mathematical structures [48]. A category consists of a set of objects, which represent types, and a set of morphisms, which define mappings among types. These mappings can include relations, conversions, mathematical functions, etc. For example, a morphism sign: R → S converts any real number in R to a value in S, S={-1,0,1} according to its sign: a positive number maps to 1, a negative number maps to -1, and zero maps to 0. Morphisms can also act on multiple objects and generate multiple objects. For example, a morphism R → S converts the sign of a product of two real numbers to a value in S.

Analogous to the mapping of concepts to models [49], architecting is a mapping from the Problem Domain to an Architecture Co-Domain. This mapping should correspond to the notion that Deciding is a mapping from Problem to Decision. Architecting and deciding can be viewed as categorically equivalent if there exists a complete mapping of the decision domain to the architecture domain. In Category Theoretic terms, a mapping between categories is a functor. We would like to show that there exists a functor ADF: A → D such that for every object and morphism in A there exists a mapping to objects and morphisms in D. Similarly, we would like to show that there exists a functor DAF: D → A such that for every object and morphism in D there exists a mapping to objects and morphisms in A.

In a category of system architectures, the objects are Architectures A, and morphisms are mappings of one architecture to another architecture, i.e. *architecting* steps. We shall initially argue simply that *architecting*: A → A. The morphism *architecting* includes a set of elaborations that change the architecture to address the problems that the previous architecture presents. This mapping also considers options to change the current architecture. Therefore, it would be more correct to state that architecting {A,P,O} → A, where P is a problem object, and O is a candidate operation object.

Each architecture A(n) presents a set of problems P, that has to be solved by the architecture A(n+1). The sequence of solving the problem may transit through a set of architectural alternatives A(n+1,1), A(n+1,2), A(n+1,3), etc. Therefore, architecting is also a polymorphism on A(n) due to its ability to create multiple sequel architectures. Alternative architectures are generated according to candidate operations on A(n). For example: add/modify/remove a block (structural element), add/modify/remove a function, add/modify/remove operand, add/modify/remove assignment of function to form, add/modify/remove output relation from function to operand, and so on. We can also merge or split items - e.g., break down a function into two or more smaller functions, merge several outputs into one big output entity, etc.

Some revisions of a given architecture model are not recommended, even if they are syntactically valid using a given modeling language. We should follow the careful transition through our concept representation framework, in order to maximize stakeholder value and solution-neutral problem definition, to ensure solution tradespace exploration, appropriately follow architecting guidelines, and minimize solution discrepancy.

While every architecting step changes the architecture, not every operation constitutes a decision problem. A decision point emerges when multiple options are possible. Although any inclusion or exclusion of an item in the architecture could pose a dilemma, constitute a decision point, or incur a discussion, we will usually conclude that it is preferable and worthwhile to include rather than exclude any aspect that enriches the concept and context of the architecture. For instance, we should not refrain from including any stakeholder or stakeholder needs even if they seem far-fetched or infeasible at a specific point in time. In case we wish to consider alternatives with and without a specific feature, capability, or aspect, we should define its state set as a binary existent/non-existent such that it will be taken into account in the definition of the tradespace. We therefore define an architecture decision point for A(n) as a situation in which all the following conditions hold:

1. at least two options are possible regarding a specific aspect of A: $A(n) \rightarrow A^1\ (n+1);\ A(n) \rightarrow A^2(n+1); \ldots$

2. Choosing one option over another may result in an architecture change, in a different solution, or in a different cost-benefit balance with respect to original stakeholder needs: $A^i\ (n+1)\ \neq\ A^j(n+1)$, for any i,j.
3. The decision cannot be made at a later point in time, i.e., the next iteration must be different from the current: $A^i(n+1)\ \neq\ A(n)$ for any i.

The above three conditions allow very specific issues to become decision points, and filter out trivial architecture modifications. However, it is now critical to identify these decision points and separate them from the rest of the architecture modification steps. Tagging model elements as decision variables enables this. For example, a sensor could be on or off from the operational perspective, but could be local or remote from the architectural perspective. Sensor activation (on/off) is obviously not an architectural decision, but determining the sensor location (local/remote) is. Therefore, we would characterize the sensor with two attributes: activation state and location. Only the location of the sensor is an architectural decision. We will tag that as a decision for further filtering and analysis.

Decision problems have been traditionally captured using decision trees [13, 50]. One of the major issues with decision trees is their obvious decision-centricity, as opposed to solution-centricity: they emphasize the decision-making process, but it is sometimes difficult to see how decisions represent solutions. Said otherwise, once a system architecture has been created and built, the architect's intent may vanish. Conversely, system architectures show the result of implemented architectural decisions, but not the decision-making process that effected those decisions.

A computational framework for coordinated design of cyber-physical system components under a given architecture (e.g. power, mass, and capacity optimization for a vehicle, communication system, etc.) attempted to address this decision-to-design discrepancy [51]. The problem is framed as the resource intake required to deliver the assigned functionality relative to an existing architecture. We are interested in extending this approach to develop a tradespace of architectures and select preferable architectures. We need to capture both the design and the designing process, in a coordinated and consistent way. This is where MBSE fits in.

Although MBSE has focused on representing system architectures with diagrams, it is possible to harness the power of modeling notations to capture decision processes. We should therefore strive to encode the process of deciding about architectural options in the conceptual system model, as well as its outcomes (in the form of architecture artifacts), this dual architecture-decision, we may begin to generate a model-based decision-driven system

architecture. This assertion has been initially corroborated in [36] where a canonical decision-making process model was introduced.

Modeling the architectural decision-making process also facilitates better planning of the process. By planning ahead multiple architecting iterations, the system architect can clarify the prioritization of needs or the preference of solution technologies along predefined milestones. The model does not have to show only the immediately decision at hand. By deriving architectures from other architectures we will be able to serialize the architectural decision-making process and significantly reduce the computational and cognitive effort that is necessary to reason about a combinatorically-exploding tradespace. In the above example, rather than consider 16 candidate combinations of the four binary decision variables, the first iteration considers 8 and the second iteration considers 2, hence the total number of reviewed candidates is 10. While this may result in overlooking 6 candidate architectures, a decision to prioritize decision variables A, B, and C and then decide about D, due to several legitimate considerations, is practically encoded in this model and can even be audited, debriefed, or even revisited - if it will not be too late.

MBSE is an environment that can foster concept discovery. In practice, reaching a viable architecture is a major milestone, short of comparing multiple architectures. MBSE environments could, at minimum, assist the architect in detecting this moment, for example by showing the architect that satisfactory coverage of needs has been achieved.

MBSE should also be able to indicate to the architect that a work-in-process architecture is infeasible or prone to reach a dead end. MBSE environments can fulfill this role if they are able to track the fulfillment of goals, whether those goals are defined within the model or as external evaluations and judgements that the model has to satisfy.

1.3.6. Solution-specific architecture decisions

Architectural decisions constitute a gateway between the problem-oriented decision point formulation of the architecture, and the solution-oriented architecture specification and elaboration. While the decisions are made based on cost and benefit considerations, we recall that the architecting morphism also accounts for the set of available operations on the given architecture model. However, the size of this set of operations is completely arbitrary. Namely - we can carry out any number of operations we feel is sufficient to establish confidence in the architecture's ability to deliver the

costs and benefits that we assess for it. In a world of incomplete information, constant change, and limited resources, we must make the call in many cases, and often end up revisiting and revising our former decisions.

MBSE could make this process of elaborating an architecture more structured. For example, we can insist on elaborating any architectural candidate one or two levels down in order to gain confidence in that potential solution. In many cases, these extra levels of detail could help us realize we are about to reach a dead end or come up with a solution that does not make sense.

Following the model-based specification of both the architecture and the architectural decision-making process, and remaining within the same conceptual modeling framework, we can explore candidate architectures by mapping them to what models truly excel in – the architecture's functional and structural specification. MBSE supports the specifying of operational processes and solution-neutral functions to be supported by the architecture, and mapping each operational process to the stakeholders that are involved in it, each operand to a need, and each functionality to the operational process that it contributes to, either by plain membership in the set of activities that compose the process, or through the specific generation of output or outcomes that can be used in the operational process.

A robust MBSE environment provides for both diverging from a problem statement to a space of alternative solutions, as we have seen so far, and for converging on a specific solution or subset of solutions as the architect desires, as we discuss next. The ability to continue using the same environment to grow a conceptual architecture into a comprehensive solution design is a major benefit, and yet, it is not always the case with MBSE environments.

A concept has to include solution-specific operands, processes, form, and the allocation of form to function, to make a good architecture. MBSE can force or advise architecture elaboration in order to capture these aspects. Robust MBSE environments will also make it easy for the architect to compare concepts both within the same model and across multiple models. The former approach helps in seeing the big picture in one place, while the latter helps in letting the solution architect focus on the details of the solution.

For driver behavior tracking, suppose we have chosen an open architecture, we can now incorporate an in-vehicle camera, an environment recording camera, a vehicle-integrated data collection device, or a driver assistance system, to collect information from the vehicle. In fact, any combination of these four solutions could be a candidate solution as well, and while some combinations may potentially provide greater value on some criteria, e.g. those that contribute to fulfilling the operational needs, they may also be too

expensive, complicated, big, heavy, etc. All of these factor into the stakeholder needs, but now we can actually start discussing such attributes as mass, volume, power consumption, bandwidth, and performance. This is where architectural modeling both informs the more technical and functional architectural decisions (e.g. where to install a camera, or how to process the data), but, as emphasized, it also helps validate the open sensor architecture that we have chosen in the conceptual architecting phase.

By mapping each solution we propose to a stakeholder need in a model, where everything is traced back to the preference schemes and framed as a decision-driven process, is again a significant game changer. It is always a challenge to come back to a stakeholder with a concept and elicit new requirements and expectations. It is also clear that stakeholders must remain in the loop because the solution directly impacts the concept of operations, as shown in the concept representation framework. Even if the conceptual architecture is converged, this phase opens up a whole new tradespace of options and decision-making remains critical and essential as it previously was in mapping out stakeholders needs to solution concepts.

Transparent, collaborative, model-based stakeholder engagement significantly improves stakeholder need validation by introducing visual, model-driven projections of solution architectures on problem statements. Choosing a vehicle-integrated solution inevitably generates concerns about interference with vehicle control and the potential risk of cyber-security.

The outcome of solution generating vis-à-vis well-defined stakeholder needs is illustrated in Figure 17. We begin with a functional decomposition of the system's main functionality, while ensuring the nested functionalities are as solution-neutral and as need-oriented as possible. In our case we specify Road Monitoring, Driver Monitoring, and Vehicle Monitoring as SSPs that map to the Monitoring SNP. Real-time Analysis and Off-Line Analysis are two solution approaches for Analysis. Informing the Vehicle Owner and the Driver can be implemented either in Real-Time or after the fact.

Specifying potential forms that may contribute to the execution of each function lays out the tradespace of possible combined solutions. We recall that any combination of options constitutes a theoretically possible solution and implies a decision point. We map each SSP to Generic Forms. We then propose three integrated architectures: minimal, maximal, and midway architectures. This is also a way to reduce the problem-space from an exponential combination to a set of functions that must be performed by the system, such that all functions and needs are fulfilled, or at least fulfillable. This

representation does not directly address the decisions to be made, but we can decide for each GF artifact whether to implement it or not.

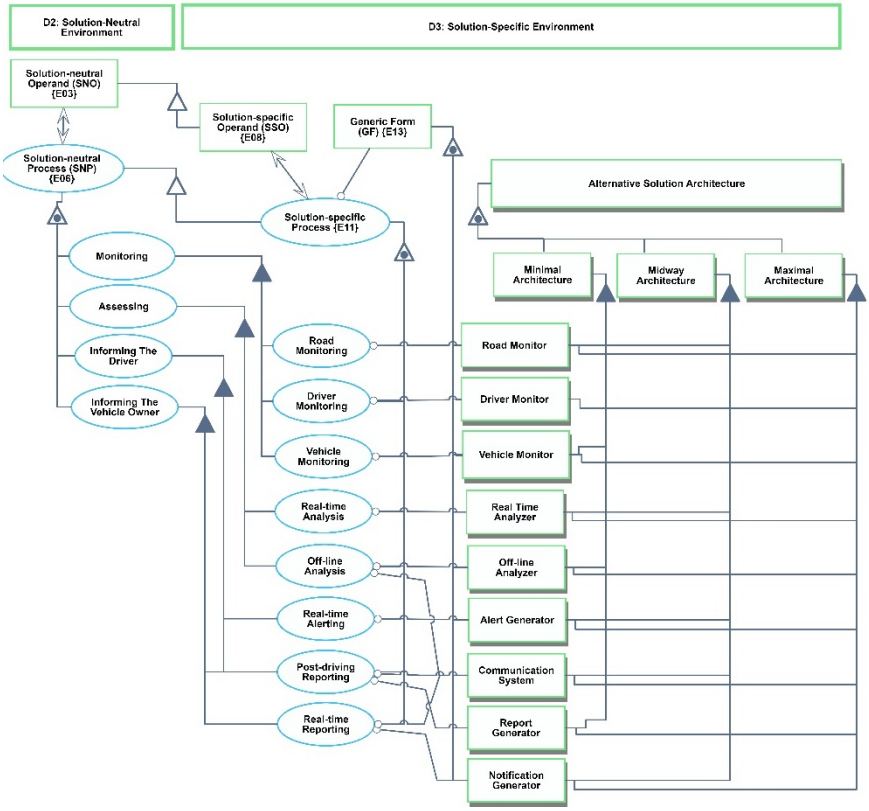


Figure 17. Deriving solution-specific functions from solution-neutral functions, specifying generic form to support solution-specific functions, and assembling three typical architectures - minimal, midway, and maximal - as combinations of generic form. The selected architecture will be further developed in D4.

We must provide at least one form to provide each function. In some cases, the same form may perform more than one function, as the technology, or commercial product it relies on allows. In some cases, a solution candidate requires another solution candidate. For instance, in our example, in order to run a software application within the vehicle, we must be able to run it either on a dedicated device or on the vehicle's multimedia system.

In this example we define a Minimal Architecture as the combination of a Vehicle Monitor, Off-Line Analyzer, and Report Generator. We can also refer to this trio as one architectural building-block and define any architecture by adding to this one. By specifying candidate elements of form and

function without specifying their states we argue that they can either exist or not exist in the solution. This is another approach to list solution factors that can later co-facilitate combined solution architectures, complementing the approach for listing options as states of the constituent decision units. With three out of nine elements unified, we can now say that the solution space has $2^7 = 128$ options. Any solution which includes more than the Minimal Solution and less than the Maximal Solution (which includes all the eight elements) is a partial solution that we can specify in terms of its constituent elements and evaluate according to our decision criteria. A model is useful here both in visualizing the range of options, and in elaborating any point in this range.

Decision criteria might include overall weight, volume, power consumption, and integration effort. Note that such design parameters may emerge as constraints to a solution rather than pre-conceived stakeholder needs, which is yet another reason for an iterative model-based architectural decision-making approach as we advocate here. We can begin discussing the cost, in financial, energetic, or performance-related terms, after we have secured a solution for stakeholder needs. Another approach would be to combine these benefit and cost factors together and consider them together in the same iteration. Both approaches are possible in an MBSE environment, as the evolvability of the model is an inherent capability of the MBSE process. As explained before, it depends on the decomposability and conceivability of the tradespace and is up to the solution architect to figure out. Physical qualities must be considered together because of the mutual effects (e.g. the combination of mass, power, and capacity). The model can clearly capture bundles of solution-specific attributes as decision variables and facilitate (and in some cases execute) the computation of a Pareto frontier, i.e. a set of combinations that meet all the criteria at the best values. We can similarly analyze software considerations and determine the most appropriate decomposition into digestible and sensible design decisions.

We may argue that the contribution of in-vehicle camera is smaller than that of a road-observing camera, or that the integration with a Driver Assistance System is better for alerting the driver than a multimedia system interface. The purpose here is not necessarily to argue for one approach in favor of another, but to show how a model-based scheme can greatly enhance the visibility of the decision-making process, the understanding of trade-offs and composition of alternatives, and the propagation of value and enablement all the way to stakeholder needs. Communicating such a model to stakeholders is also easier and more intuitive, as it can illustrate the impact on stakeholder needs, which greatly increases clarity and transparency.

1.4. Conclusion

We have shown how systems architecting is essentially a reasoning process, which consists of conceptualization and decision-making steps. MBSE facilitates, but also changes the architecting process. It is worth taking a step back to summarize the ways in which MBSA differs and grows from legacy, "off-line" system architecting. The architecting process (whether legacy or model-based) is a source of discovery and insight. It addresses the need to understand, through sufficient specification and completeness of coverage at a given abstraction level of analysis. These drivers enable us to judge when the architectural modeling is concluded.

We have presented a model-based concept representation framework that formalizes the conceptualization process to generate a system architecture. The framework advocates a path to a solution that accounts for all the major concerns in a system architecture. Indeed, this is not a linear process, but an iterative one, rife with revisions, divergence, and convergence. The framework accommodates both ongoing conceptualizations and decision-making when critical decisions must be made.

We conclude with three key MBSA principles:

1. MBSA is an iterative reasoning process, in which the model records and informs the evolving conceptual architecture.
2. MBSA fosters divergence before convergence: allowing for options to emerge from a solution-neutral environment, and converging on a solution after considering multiple solution-specific approaches.
3. MBSA projects can focus on the relevant conceptualizations and not necessarily follow an A-to-Z approach - the amount of modeled information should be just enough to reach a decision.

The "offline" architecting process is based on siloed analysis and discussion, and we find there is substantially more effort in setting up a model to answer architectural questions. Some of this effort is due to the need to translate decision-theoretic concepts into conceptual modeling language. However, once a model is developed and can serve as a baseline, the resulting exploration becomes quicker with each iteration. The model-based approach could provide a return on investment due to the reusability and evolvability of model assets through multiple iterations. We caution, though, that the architecting process should not be conceived as a procedural one: our framework is not just a checklist. Given that no architectural model will be able to capture detailed design information, we find that the process of modeling

and sense-making is important to building a shared understanding among the architecture team members. Therefore, the process remains both cognitive and computational, and the two streams augment each other thanks to the shared artifact that the model constitutes, which lends itself both to human reasoning and machine processing.

We summarize the primary benefits of MBSA as follows:

Communication with stakeholders: Communicating with stakeholders is an essential part of the decision-making process. Engaging stakeholders, getting them involved, committed, and informed, is critical for architecting the solution to their needs. A shared language for discussing concepts, problems, options, and solutions with a reference model is instrumental in facilitating, formalizing, and documenting the discussion.

Knowledge and architectural building block reusability: A growing concern in enterprises working on evolving systems in an agile world, the reuse of existing knowledge about the domain and design of existing designs, is becoming a critical aspect of MBSE. Reusability of knowledge, architecture, and design artifacts has two major impacts on the decision-making process. Reliance on existing assets reduces uncertainty, ambiguity, and programmatic risk (schedule, budget, quality, etc.). Furthermore, reducing the explorable tradespace by reusing existing component designs is an approach to coping with a combinatorial solution space explosion. System architecture decisions that are based on a formal model that incorporates reusable domain content and building blocks have higher degrees of confidence and may be more likely to survive the emergence of new constraints, issues, or materializations of risk.

Consistent problem-Space mapping: MBSA facilitates conceptual mapping of the problem space in a domain-agnostic manner. It is then possible to specify problem-domain needs, key evaluation metrics, and performance indicators, and to layout the architectural solution space that may contain at least one feasible solution to the problem. Evolving problem understanding, some of it in parallel to decision-making, helps reshape and recreate the problem domain. We believe that this consistency in mapping the problem space helps us recognize patterns across designs.

Gradual transformation of problem statements to architectures: MBSA facilitates a smooth transition from the problem domain to the solution domain by allowing for a traceable mapping of candidate and chosen solutions to problems. While we could describe the problem domain in a variety of less-formal ways, and the solution in a variety of approaches, a

model-based approach formalizes the transition, the transformation, and the traceability of solution domain aspects to problem domain ones.

Consistent comparison and evolution of alternative architectures: Comparing and reasoning about architectures within the modeling frameworks is challenging. MBSA allows this by mapping conceptual architecture decisions to quantifiable metrics of stakeholder utility on the one hand, and to quantifiable metrics of design validity on the other. We can promise a conceptual architecture that helps our stakeholders in many ways, but failing to validate the architecture may be bad for business. MBSA facilitates a smooth transition across modeling, assessment, verification and validation methods, with a conceptual model as a focal point. The alternative approach rephrases the decision-making problem in mathematical decision-analytic terms that do not rely on the model. There are two risks there: a) the full architecture scope will not be sufficiently understood when developing a selected solution, and b) validating the solution architecture by tracing back to the "numbers" will be impossible.

Documenting both the architecting process and its outcome—the architecture: By documenting our decision-making process in a model, we are creating a mapping of the entire process rather than the outcome. One should not guess why a particular architecture has been selected. We believe MBSE will be much stronger in capturing functional intent. A decision-oriented model can substantiate the solution on the considerations that fed the decision-making process leading to that particular solution. While this is not a common MBSE practice, we assert that it is possible, achievable, and desirable within an MBSE framework to ensure that the process is appropriately documented, and that the traceability of solutions to options to problems to needs to stakeholders becomes clear.

Aspiration to sufficiency: We have discussed the importance of model sufficiency – knowing that we have modeled enough to make an architectural decision. We reasoned that by applying Miller's Law of ~7±2 elements in every set as a threshold for sufficiency. We can measure the compliance of various sets of objects and processes with this criterion and determine our confidence in the model and our willingness to rely on it.

Acknowledgement. We would like to thank the MIT-Technion Post-Doctoral Fellowship Program for enabling this collaboration.

1.5. References

1. Dori D (2016) Model-Based Systems Engineering with OPM and SysML. Springer, New York
2. McDermott TA, Hutchinson N, Clifford M, Van Aken E, Slado A, Henderson K (2020) Benchmarking the Benefits and Current Maturity of Model-Based Systems Engineering across the Enterprise. Systems Engineering Research Center (SERC)
3. Hale JP, Zimmerman P, Kukkala G, Guerrero J, Kobryn P, Puchek B, Bisconti M, Baldwin C, Mulpuri M (2017) Digital Model-based Engineering: Expectations, Prerequisites, and Challenges of Infusion. NASA
4. Morris BA, Harvey D, Robinson KP, Cook SC (2016) Issues in Conceptual Design and MBSE Successes: Insights from the Model-Based Conceptual Design Surveys. INCOSE Int Symp 26:269–282 . https://doi.org/10.1002/j.2334-5837.2016.00159.x
5. Weilkiens T, Lamm JG, Roth S, Walker M (2016) Model-Based System Architecture. In: Model Based System Architecture, First Edit. John Wiley & Sons, Inc, pp 27–33
6. Object Management Group (2019) OMG Systems Modeling Language Version 1.6
7. Klappholz D, Port D (2004) Introduction to MBASE (Model-Based (System) Architecting and Software Engineering). In: ZELKOWITZ M V. (ed) Advances in Computers. Elsevier, pp 203–248
8. Boehm B, Klappholz D, Colbert E, Puri P, Jain A, Bhuta J, Kitapci H (2004) Guidelines for Model-Based (System) Architecting and Software Engineering (MBASE). 1–159
9. Boehm B (2006) Some future trends and implications for systems and software engineering processes. Syst Eng 9:1–19 . https://doi.org/10.1002/sys.20044
10. Boehm B, Oram A, Wilson G (2010) Architecting: How much and when? O'Reilly Media
11. Bahill AT, Henderson SJ (2005) Requirements Development, Verification, and Validation exhibited in famous failures. Syst Eng 8:1–14 . https://doi.org/10.1002/sys.20017
12. Lane JA, Koolmanojwong S, Boehm B (2013) Affordable Systems : Balancing the Capability , Schedule , Flexibility , and Technical Debt Tradespace
13. Crawley E, Cameron B, Selva D (2015) Systems Architecture: Strategy and Product Development for Complex Systems. Prentice Hall
14. Bahill AT, Madni AM (2017) Tradeoff Decisions in System Design. Springer International Publishing Switzerland
15. Dori D, Kohen H, Jbara A, Wengrowicz N, Lavi R, Levi-Soskin N, Bernstein K, Shani U (2020) OPCloud: An OPM Integrated Conceptual-Executable Modeling Environment for Industry 4.0. In: Kenett RS, Swarz

RS, Zonnenshain A (eds) Systems Engineering in the Fourth Industrial Revolution: Big Data, Novel Technologies, and Modern Systems Engineering. Wiley
16. Menshenin Y, Mordecai Y (2020) Model Based System Architecting Reference Model. V01_20_12
17. Chomsky N (1956) Three models for the description of language. IRE Trans Inf Theory 2:113–124 . https://doi.org/10.1109/TIT.1956.1056813
18. INCOSE (2015) INCOSE Systems Engineering Handbook: A Guide for System Life Cycle Processes and Activities, Fourth Edi. John Wiley & Sons, Inc.
19. Menshenin Y, Crawley E (2020) A system concept representation framework and its testing on patents, urban architectural patterns, and software patterns. Syst Eng 23:492–515 . https://doi.org/10.1002/sys.21547
20. Menshenin Y (2020) Model-based framework for system concept - Ph.D. Thesis. Skolkovo Institute of Science and Technology
21. Freeman RE (2001) A Stakeholder Theory of the Modern Corporation. Perspect Bus Ethics 3: . https://doi.org/10.3138/9781442673496-009
22. European Commission (2019) The European Green Deal. Brussels
23. NASA (2016) NASA System Engineering Handbook, SP-2016-61. NASA
24. Suh NP (1990) The principles of design. Oxford University Press on Demand
25. Nordlund M, Lee T, Kim S-G (2015) Axiomatic Design: 30 Years After. In: Proceedings of the ASME 2015 International Mechanical Engineering Congress and Exposition IMECE2015. ASME, Houston, Texas
26. Pahl G, Beitz W, Feldhusen J, Grote K-H (2007) Engineering Design A Systematic Approach. Springer-Verlag London
27. Maier JF, Eckert CM, Clarkson PJ (2016) Model granularity and related concepts. In: Proceedings of the DESIGN 2016 14th International Design Conference
28. Eppinger SD, Browning TR (2012) Design Structure Matrix Methods and Applications. Des Struct Matrix Methods Appl. https://doi.org/10.7551/mitpress/8896.001.0001
29. Miller GA (1956) The magical number seven, plus or minus two: Some limits on our capacity for processing information. Psychol Rev 63:81–97 . https://doi.org/https://doi.org/10.1037/h0043158
30. United States Department of Defense (DoD) (2010) The DoDAF Architecture Framework Version 2.02. https://dodcio.defense.gov/Library/DoD-Architecture-Framework/. Accessed 3 Dec 2020
31. Mordecai Y, James NK, Crawley EF (2020) Object-Process Model-Based Operational Viewpoint Specification for Aerospace Architectures. IEEE Aerosp Conf Proc 1–15 . https://doi.org/10.1109/AERO47225.2020.9172685
32. Maier MW, Rechtin E (2000) The Art of Systems Architecting, Second Edi. CRC Press LLC
33. Cambridge Dictionary (2020) Decision.

https://dictionary.cambridge.org/us/dictionary/english/decision. Accessed 18 Dec 2020
34. Zeleny M (1982) The Decision Process and Its Stages. In: Zeleny M, Cochrane J (eds) Multiple criteria decision making. McGraw-Hill, Inc., New York, pp 85–95
35. Weinreich R, Groher I (2016) The Architect's Role in Practice: From Decision Maker to Knowledge Manager? IEEE Softw 33:63–69 . https://doi.org/10.1109/MS.2016.143
36. Mordecai Y, Dori D (2014) Conceptual Modeling of System-Based Decision-Making. In: INCOSE Internaional Symposium. INCOSE, Las-Vegas NV, USA, USA
37. Pratt, Raiffa, Schlaifer (1964) The Foundations of Decision Under Uncertainty. 59:353–375
38. Saaty TL (1990) How to make a decision: The analytic hierarchy process. Eur J Oper Res 48:9–26 . https://doi.org/10.1016/0377-2217(90)90057-I
39. Howard R (1968) The Foundations of Decision Analysis. IEEE Trans Syst Sci Cybern 4:211–219 . https://doi.org/10.1109/TSSC.1968.300115
40. Kahneman D (2003) A perspective on judgment and choice: mapping bounded rationality. Am Psychol 58:697–720 . https://doi.org/10.1037/0003-066X.58.9.697
41. Tversky A, Kahneman D (1974) Judgement under Uncertainty: Heuristics and Biases. Science (80-) 185:
42. INCOSE (2015) INCOSE Systems Engineering Handbook: A Guide for System Life Cycle Processes and Activities, Fourth Edi. John Wiley & Sons, Inc., San Diego, CA, USA
43. Haskins C, Forsberg K, Krueger M, Walden D, Hamelin RD (2011) Systems Engineering Handbook, v. 3.2.2. International Council on Systems Engineering
44. Parnell GS, Parnell GS, Madni AM, Bordley RF (2017) Trade-off Analytics : Creating and Exploring the System Tradespace Chapter 2 : A Conceptual Framework and Mathematical Foundation for Trade-off Analysis
45. Rebentisch ES, Crawley EF, Loureiro G, Dickmann JQ, Catanzaro SN (2005) Using Stakeholder Value Analysis to Build Exploration Sustainability. Engineering 1–15 . https://doi.org/10.2514/6.2005-2553
46. Malak RJ, Aughenbaugh JM, Paredis CJJ (2009) Multi-attribute utility analysis in set-based conceptual design. CAD Comput Aided Des 41:214–227 . https://doi.org/10.1016/j.cad.2008.06.004
47. Ross AM, Hastings DE, Warmkessel JM, Diller NP (2004) Multi-Attribute Tradespace Exploration as Front End for Effective Space System Design. J Spacecr Rockets 41:20–28 . https://doi.org/10.2514/1.9204
48. Breiner S, Sriram RD, Subrahmanian E (2019) Compositional Models for Complex Systems
49. Mordecai Y, Fairbanks J, Crawley EF (2020) Category-Theoretic Formulation of Model-Based Systems Architecting : The Concept →

Model → Graph → View → Concept Transformation Cycle
50. Haimes YY (2009) Multiobjective Decision-Tree Analysis. In: Risk Modeling, Assessment, and Management, Third Edit. John Wiley & Sons, Inc.
51. Censi A (2017) A Class of Co-Design Problems with Cyclic Constraints and Their Solution. IEEE Robot Autom Lett 2:96–103 . https://doi.org/10.1109/LRA.2016.2535127

1.6. Cross-References

TBD

1.7. Author Bio (for static eReference)

Yaroslav Menshenin, PhD, is a Research Scientist at the Space Center, Skolkovo Institute of Science and Technology – Skoltech (Moscow, Russia). He holds a PhD (2020) from Skoltech and a Specialist Degree (MSc equivalent) (2012) from the National University of Science and Technology "MISIS" (Moscow, Russia). He is also a graduate of the Singularity University located at NASA Ames Research Center (California, USA). Dr. Menshenin is a Member of INCOSE, AIAA, IFIP WG 5.1, and the DESIGN Society. He was also a visiting doctoral candidate at the System Architecture Group, MIT (2016-2017).

Yaniv Mordecai, PhD, is a post-doctoral research fellow at the Engineering Systems Laboratory, Massachusetts Institute of Technology (Cambridge, Massachusetts, USA). He holds a PhD (2016) from Technion – Israel Institute of Technology, (Haifa, Israel); and MSc (2010) and BSc (2002) from Tel-Aviv University (Tel-Aviv, Israel). He is also a senior systems architect with Motorola Solutions. Dr. Mordecai is a senior member of IEEE and board member of IEEE Israel and of the Israeli Association for Systems Engineering – INCOSE_IL. Yaniv is the recipient of the IEEE Systems, Man, and Cybernetics Society Doctoral Dissertation Award (2017) and the OmegaAlpha Association Exemplary Doctoral Dissertation Award (2017).

Edward F. Crawley, ScD, is the Ford Professor of Engineering and Professor of Aeronautics and Astronautics and Engineering Systems at Massachusetts Institute of Technology (Cambridge, Massachusetts, USA). He holds a Sc.D (1980), M.S. (1978), and B.S. (1976) from MIT. Dr. Crawley was the founding president of Skolkovo Institute of Science and Technology, Moscow (2011-2016), Director of the Bernard M. Gordon – MIT Engineering Leadership Program (2007-2012), Executive Director the Cambridge University-MIT joint venture (2003-2006), and head of the Aeronautics and Astronautics Department at MIT (1996-2003). He is a Fellow of AIAA and RAeS; Member of the National

Academy of Engineering (NAE), Royal Academy of Engineering (RAEng UK), Royal Swedish Institute of Engineering Science (IVA) and the Chinese Academy of Engineering (CAE); NASA Astronaut Finalist (1980); and Regional Soaring Champion (1991, 1995, and 2005).

Bruce G. Cameron, PhD, is the Director of the System Architecture Group, Lecturer in System Design and Management, and Faculty Director of the Architecture and Systems Engineering Certificate Program – all in Massachusetts Institute of Technology (Cambridge, Massachusetts, USA) He holds a PhD (2011) and dual MS (2007) from MIT and a B.A.Sc from the University of Toronto (Toronto, Canada). Dr. Cameron is also a co-founder of Technology Strategy Partners.