26th Annual INCOSE International Symposium (IS 2016)
Edinburgh, Scotland, UK, July 18-21, 2016

Simplifying (and enriching) SysML to perform functional analysis and model instances

Jean-Luc Voirin
Thales Airborne Systems
Thales Technical Directorate
jean-luc.voirin@fr.thalesgroup.com

Stéphane Bonnet
Thales Corporate Engineering
stephane.bonnet@thalesgroup.com

Daniel Exertier
Thales Corporate Engineering
daniel.exertier@thalesgroup.com

Véronique Normand
Thales Technical Directorate
veronique.normand@thalesgroup.com

Abstract. Confronted with real life and full-scale operational deployment, model-based engineering languages and tools may fail to address the wide variety of practices situations can require. This paper describes issues encountered with SysML in the context of systems architectural design. It gives an overview of the adapted language and practices that form the foundations of Arcadia and Capella, a field-proven model-based engineering method and its associated open source workbench. Relying on concrete examples, this paper focuses on modeling constructs supporting functional analysis and addresses the problematics of modeling at instance level.

Introduction

Model-based systems engineering (MBSE) is the formalized application of modeling to support system requirements, design, analysis, verification and validation activities, beginning in the conceptual design phase and continuing throughout development and later life cycle phases. While MBSE is expected to become the norm for systems engineering execution in the next ten years (INCOSE, 2014), modeling is not a guaranty of success.

Numerous modeling languages have been available for decades. These languages can be classified based on their scope of purpose, level of semantic formalization, scope of distribution, degree of standardization, their adaptation to users' cultural background, and the existence of an ecosystem (Aracic and Roques, 2010).

Successful implementations of MBSE approaches are typically the result of several factors, the first of which being a clear definition of modeling objectives. Different objectives likely mean different means: language, size of effort, organization, tools, etc.

In 2005, Thales identified MBSE as a key lever for engineering performance improvement and initiated an ambitious rollout program, investing massively on both methodological and tooling aspects. Nearly ten years later, the results include a model-based engineering method – Arcadia – and a supporting modeling workbench – Capella – deployed on various domains in all Thales Business Units worldwide and made open source in 2014 (Capella, 2014).

Arcadia and Capella primarily focus on architectural design – justification of components/interfaces through functional analysis, architecture non-functional early evaluation, and preparation of integration and validation activities – excluding low-level

behavioral modeling or simulation. This paper explains a few choices integral to development of Arcadia and Capella, based on both experimentations and system engineers' feedback. In particular, it describes the original means it provides to perform functional analysis and manage type-instances, which are different from SysML[1] approaches.

Arcadia and Capella, quick introduction

Arcadia (Voirin, 2010; Arcadia, 2014) is a model-based method devoted to systems, software, hardware architecture engineering. It describes the detailed reasoning to understand the real customer need, define and share the product architecture among all engineering stakeholders, early validate its design and justify it, ease and master integration, validation, verification (IVV).

Arcadia can be applied to complex systems, equipment, software or hardware architecture definition, especially those dealing with strong constraints to be reconciled (cost, performance, safety, security, reuse, consumption, weight…). It is intended to be embraced by most stakeholders in system/product/software/hardware definition, and by IVV actors, as their common engineering reference.

Arcadia has been experimented and validated in many real life contexts for several years now, in most Thales operational units. Its large adoption in many different engineering contexts witnesses of an industry-proven comprehensive method for system engineering, adapting to each context in a dedicated manner, and yet being tooled by the same powerful tools capitalizing knowledge.

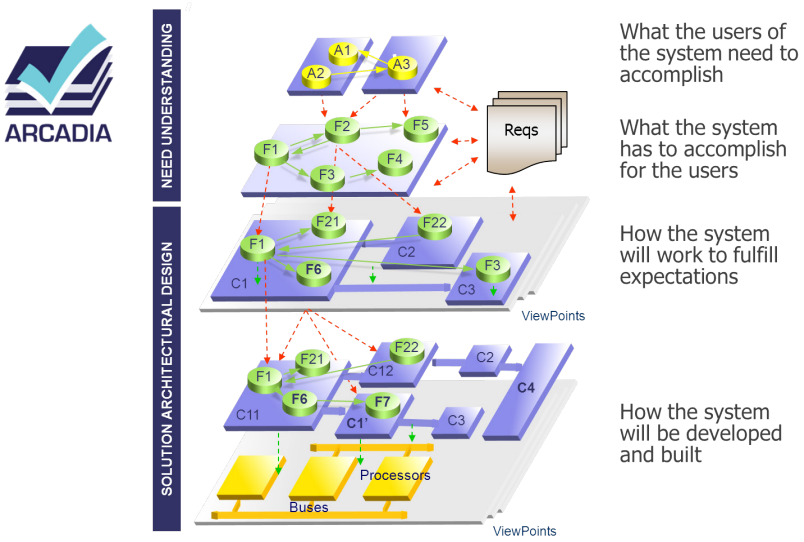

Figure 1: Arcadia engineering phases

Arcadia intensively relies on functional analysis. It introduces several engineering phases and promotes a clear distinction between the expression of the need and the expression of the solution.

The field-proven modelling workbench Capella has been developed both to guide users in applying the Arcadia method and to assist them in managing complexity of systems design with automated simplification mechanisms. A model is built for each Arcadia engineering step.

[1] SysML: Systems Modeling Language

All of these models are related by justification links and are processed as a whole for impact analysis. Arcadia is now partially published and a full publication is on its way. Capella is available as open source software.

Capella is not a SysML tool. Because it targets a wide variety of domains, it cannot be considered a Domain Specific Modeling Language (DSML) either. Instead, Capella is a hybrid approach: it is strongly inspired by SysML, that it simultaneously simplifies, modifies, and enriches.

The original audience for the Arcadia/Capella solution is primarily System engineering teams confronted to the "wall" of complexity. Their close involvement in the solution specification and maturation has led to a workbench which capabilities focus on helping design better architectures through:

- An embedded methodology browser
- Advanced mechanisms to manage complexity through computed simplifications and abstractions
- Productivity tools including model-to-model transformations, libraries of replicable elements, system to subsystem automated transition, etc.
- An ability to extend and/or specialize the core environment with add-ons addressing particular engineering concerns (e.g. performance, safety, cost, mass, product line, etc.) and carry out multi-criteria analyses of target architectures to find the best trade-offs.

While hundreds of Thales engineers have been using Capella as their main daily design workbench for a few years already, the Clarity consortium now aims at building an ecosystem around Capella (Clarity Consortium, 2015). Several major industrial organizations, tool providers, and consulting organizations have already joined the consortium.

The following sections elaborate on two significant differences between Arcadia/Capella and SysML solutions. The approaches described in this paper only reflect the Thales experience in the field of system architectural design. Because the scope of an MBSE method has direct consequences on the tooling, they should not be taken as universal, one-size-fits-all solutions.

Support of functional analysis

Functional analysis addresses the activities that the system must achieve to produce its desired outputs (Sage and Rouse, 1999). It allows translation of functional requirements into a formalized set of system functions having well-defined inputs and outputs. Functional analysis is a cornerstone in the Arcadia method. Therefore, great attention was paid to get the best adaptation to operational needs in large scale, complex programs and organisations.

Concepts and rules

Several lessons learnt from real life applications are described in (Voirin, 2012). That paper lists most frequent approaches for functional analysis building in Thales units, and explains why many traditional, top-down, delegation-based approaches and tools appear to be unable to support all these different lifecycles. Figure 2 summarizes five operational scenarios used by Thales units, many of them mixed during one project. It is equally important to be able to refine an existing function than to be able to group existing ones to provide a higher level of abstraction.

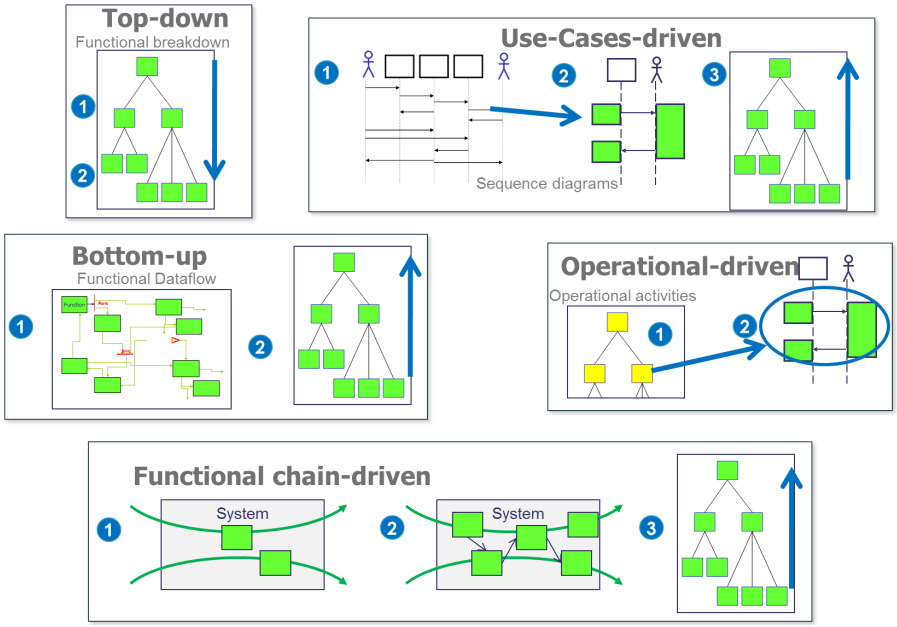

Figure 2: Five different approaches to functional analysis (functions in green, numbers give the order of modelling tasks)

Rules and concepts for functional analysis in Arcadia are meant to be compatible with these different building processes for functional analysis:

- Definition of input and output ports on functions, expressing the contracts of the function, only in terms of function production/consumption.
- Functions are linked to one another, forming a dependency graph that only formalizes functional dependencies between functions. These dependencies are expressed by oriented functional exchanges connecting the function ports.
- Nature of data, information, signals, and flows exchanged between functions is specified on both exchanges and ports.
- Definition of functions, functional exchanges, and ports is shared among all uses and diagrams.
- The functional decomposition does not rely on port delegations between parent and children functions.

This description of functional dependency graph is essential in architecture definition, because it carries the major engineering items that can define and justify architecture components contents, and their interface definition. Arcadia recommends to first create functional flows independently from components architecture, then to allocate functions to components, and finally to deduce components interfaces from functional exchanges and their contents (Voirin, 2010).

Functional decomposition

While being one of the most established modeling techniques in systems engineering, functional analysis is actually not strictly supported by SysML, which does not define the

concepts of "functions" and "function hierarchy". Function-oriented approach can however be implemented in SysML (OMG, 2012):

- Functions modeled as actions/activities diagrams. SysML actions/activities are semantically close to Arcadia functions as they are behavior elements – identified by verbs – intended to be allocated to structural elements
- Functions modeled as blocks (Lamm and Weilkiens, 2010). The functional tree is captured as a hierarchy of blocks with dataflow being represented as Internal Block Diagrams.

None of these two options met the Arcadia requirements of simplicity and scalability, as exhaustively detailed in (Voirin, 2010). Activity diagrams come with a multitude of complex constructs that are far beyond the needs of a simple Arcadia function hierarchy: activities, numerous different kinds of actions, pins, parameters, nodes, etc. Using blocks to model functions appears semantically wrong; it does not enforce the conceptual difference between structural elements and functions.

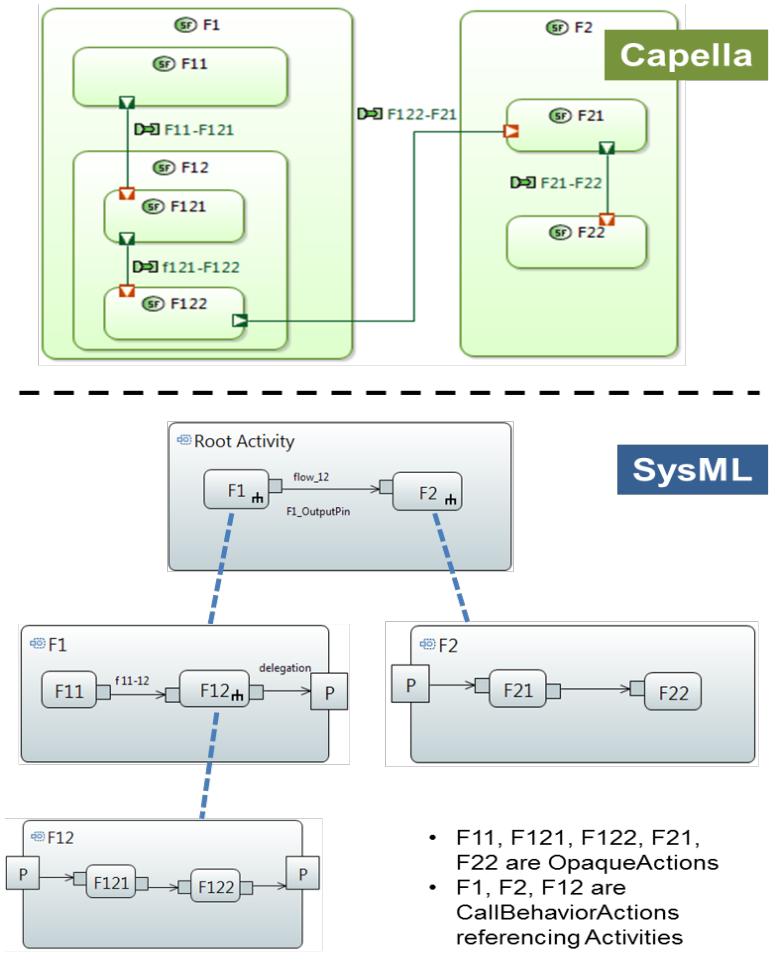

Figure 3: Equivalent functional decompositions in Capella and SysML

The main obstacle to rely on SysML as a support to Arcadia functional analysis is the encapsulation mechanism exploited in both cases to support nested functions. Figure 3 describes the fundamental difference between SysML activity diagrams and Capella functional

models. It shows how a simple functional breakdown can rapidly become complex with SysML,

In SysML activity diagrams, object flows can only be created between functions at the same level. This means that two leaf functions in the hierarchy can only communicate through delegation constructs going through their respective owning activities. Maintaining the consistency of object flows across several levels of decomposition is a tedious and error-prone task that seriously jeopardizes scalability.

In Arcadia functional models, only leaf functions can ultimately have input and output object flows, own ports, and be allocated to structural elements. Ports and object flows appearing on non-leaf functions either reflect an intermediate design that is not yet finalized, or are a computed synthesis. In the Capella workbench, the ports owned by children functions can be artificially displayed on parent functions. This makes the production of synthetic views possible at no cost (Figure 4).

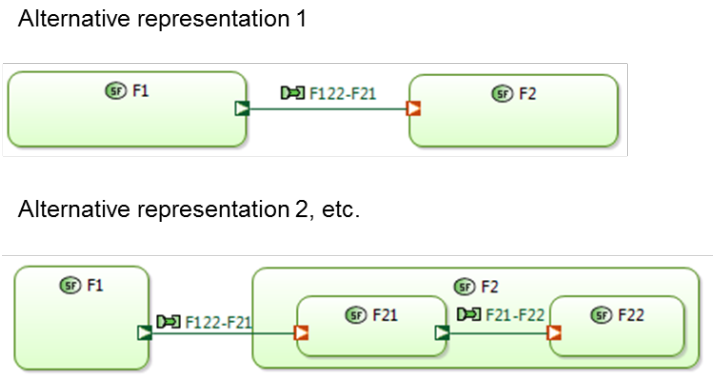

Figure 4: Computed synthetic views

The Capella implementation is well adapted to approaches described by Figure 2. Refinement work consists in creating sub functions and drag-dropping existing ports. Bottom-up approaches simply consists of grouping leaf functions in parent ones and relying on the automated production of synthetic views.

Specific focus on sequence and control flows

Beyond the functional dependency and flow-oriented views, there might sometimes be a need to describe in which order functions have to be executed. In these cases, there is a great temptation to either use functional exchanges of the dependency graph to express this sequencing order, or to add a new kind of sequence link between functions, in functional dependency graphs.

This is acceptable if the following conditions are met:

- There is a real exchange (of data, information, event, material…) between the ordered functions. Without such an explicit exchange, there is no architectural means (i.e. no reflection on interfaces) to ensure this kind of synchronization between the functions.
- This exchange is clearly identified as “activating” or “triggering” the target function (a kind of event, or request…).

- The global behavior consistency is ensured and described, especially taking into account other inputs and outputs of the ordered functions.

Other kinds of purely sequencing links, not associated to any exchange, raise problems. In the scope of architecture modeling as covered by Arcadia, this is identified as a wrong practice: data/control flow and sequence links do not have the same use and should not be mixed into the functional dependency graph.

In SysML, the semantics of control flows implies that control tokens that are transmitted from one activity to another (OMG, 2012). Because this concept of token is somehow implicit and not materialized in models, the risk of seeing these control flows misused (i.e. used as purely sequencing links) cannot be ignored. To ensure that dependency graphs are truly based on information exchanges, Capella does not provide the exact equivalent of SysML control flows between activities.

A sample scenario. These principles are illustrated hereunder with the very simple example of a vehicle service checkup: Mr. Jones and Mr. Smith come to the repair station for a checkup, at the same time.

Mr. Jones' vehicle is taken over as soon as Mr. Jones arrives: engine is hot, so it is more efficient to start by changing the oil (supposed to be done when hot and thus more fluid), before checking coolant (which requires a cold engine).

```
Hot engine scenario:
   1. Change the oil
   2. Wait for engine to be cold
   3. Check coolant
```

Mr. Smith' vehicle is taken over after Mr. Jones' one is processed: engine is cold, so it is more efficient to start by checking coolant, then to heat engine, so as to be able to change oil.

```
Cold engine scenario:
   1. Check coolant
   2. Warm engine up
   3. Change the oil
```

The way to build the associated functional analysis can be good or not, depending on either considering real data/control flow as defined above (i.e. pure functional dependencies between functions), or corrupting the data flow with (contextual) misused flows representing sequences: three different examples of the misusage of dependency graphs are discussed below.

An erroneous dependency graph. The use of dependency graph to express ordering is wrong. First, because the order is not necessarily always the same: Mr. Jones and Mr. Smith scenarios, as represented in Figure 5, appear contradictory. The dependency graph should only describe dependencies between functions that are always necessary, not occasional. In contrast, sequencing order is mostly contextual (and sequence diagrams and functional chains are meant to reflect that).

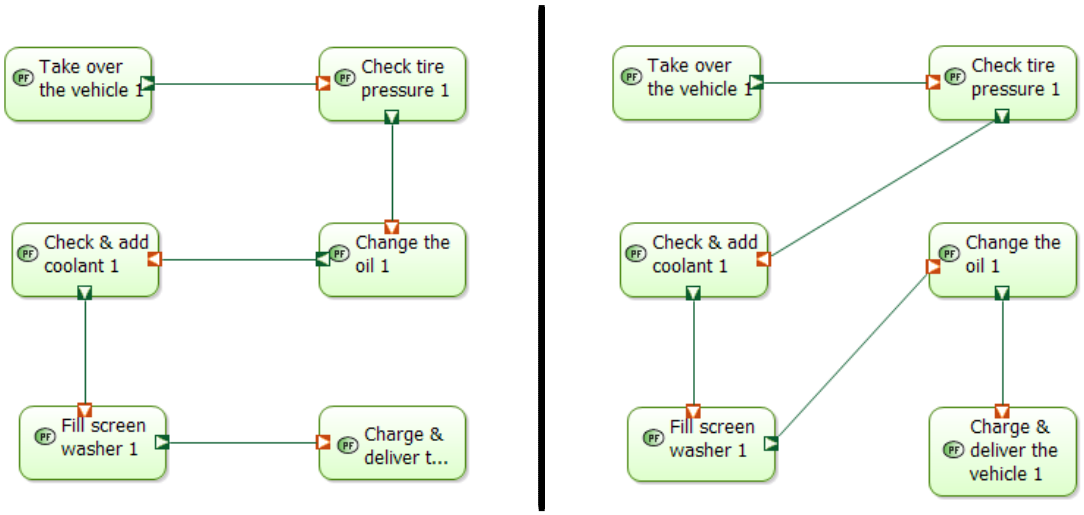

Figure 5: Erroneous dependency graph, confused with a (contextual) sequence order: two different (and incorrect) uses of dependency graphs for the same expected functions

Second, from a functional point of view, how does this view help define functional expectations?

- What could the "*Change the oil*" function need as an input, coming from *"Check tire pressure*"? And from "*Check tire pressure*" point of view, how to justify this output?
- More problematic, this view tends to hide the real input needs of each function: as an example, "*Change the oil*" requires a hot engine, so would need temperature (or clearance) as an input; similarly, "*Check & add coolant*" requires a cool engine. When focusing on sequence order, this analysis tends to be forgotten.

Improving the dependency graph. In order to improve the dependency graph, input needs and output capabilities of each function can be specified. Linking them to one another accordingly let a correct dependency graph emerge (Figure 6, information exchanges added).

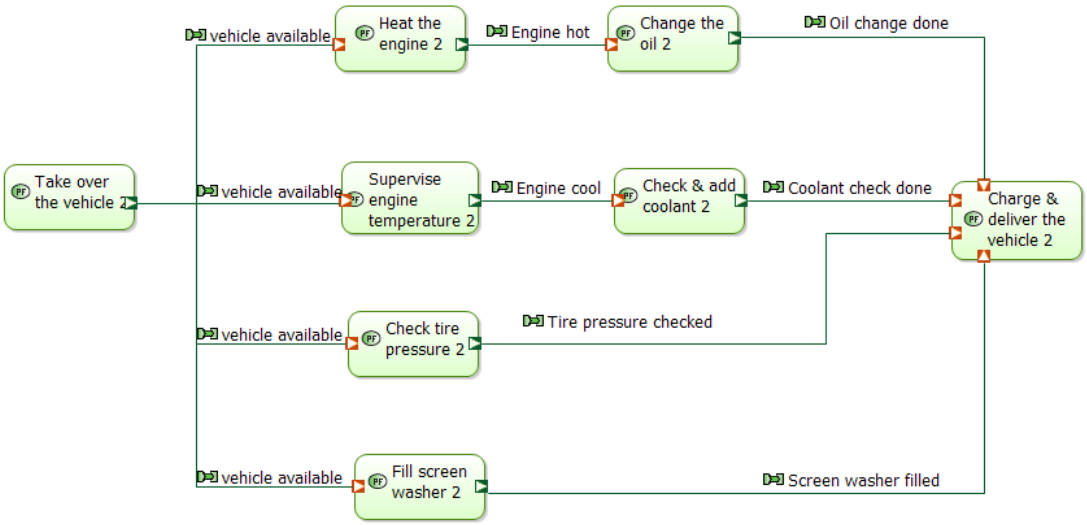

Figure 6: A correct dependency graph, with enhanced function definition and no artificial ordering links

This vision is better than the former one from a functional point of view. Dependencies are expressed in terms of data or control dependencies representing actual information exchanges between the functions. The better definition of each function inputs and outputs explicitly shows that there is no predefined order for the operations. However, modeling a proper data dependency graph is no guarantee of good design. Among others, this scheme leads to heating and cooling the engine systematically, which is not necessary in any situation, depending on order of operations. Therefore, the dependency graph as depicted by Figure 6 restricts possibilities and harms efficiency.

A much better dependency graph. The third version illustrated by Figure 7 corrects former flaws:

- Each function clearly shows what it needs to perform;
- Dependencies are correctly expressed, only based on what each function requires or is able to deliver;
- There are no contextual sequence constraints, such as ordering functions execution

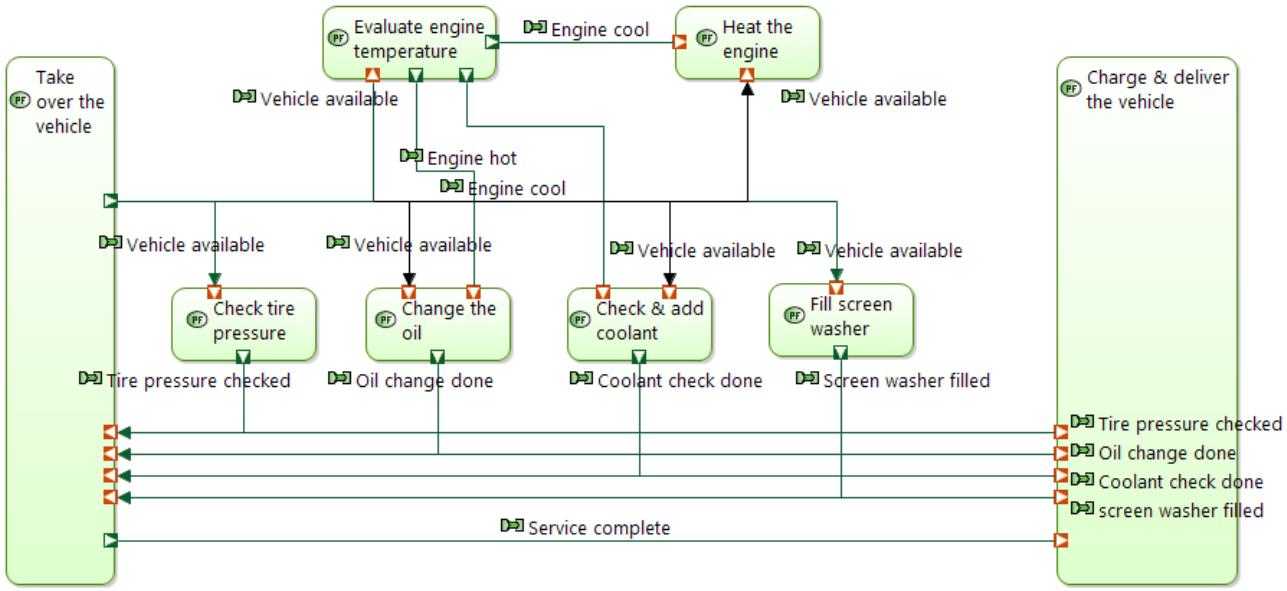

Figure 7: A more realistic and useful dependency graph

Expressing contextual ordering. Once the above pure dependency graph is correctly defined, different situations can be described with an emphasis on ordering concerns. This can be achieved by means of sequence diagrams or others formalisms (eFFBD-like[2], or even SysML activity diagrams), provided that:

- They are clearly associated to one use case in a given context, and not described as immanent.
- They separate real flows and pure sequence links, with different concepts / notations.
- No pure sequence link is involved between two functions that are allocated to two different architecture components. If this rule is transgressed, then there will be no real

[2] Extended Functional Flow Block Diagram

means to ensure the ordering. Should this need appear, a real data dependency flow would have to be defined between the two functions.

In the vehicle checkup example, both scenarios could be described in Arcadia/Capella, each in its context, as shown in Figure 8 and Figure 9 (here, sequence diagrams lifelines represent functions instead of blocks).

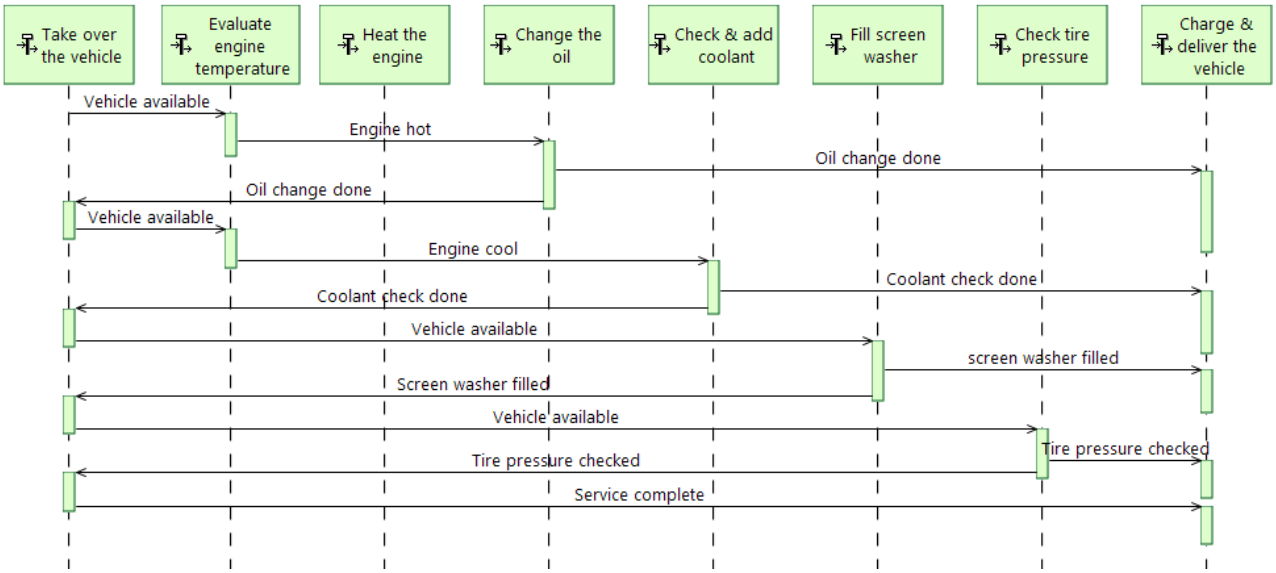

Figure 8: Mr. Jones' vehicle checkup with hot engine

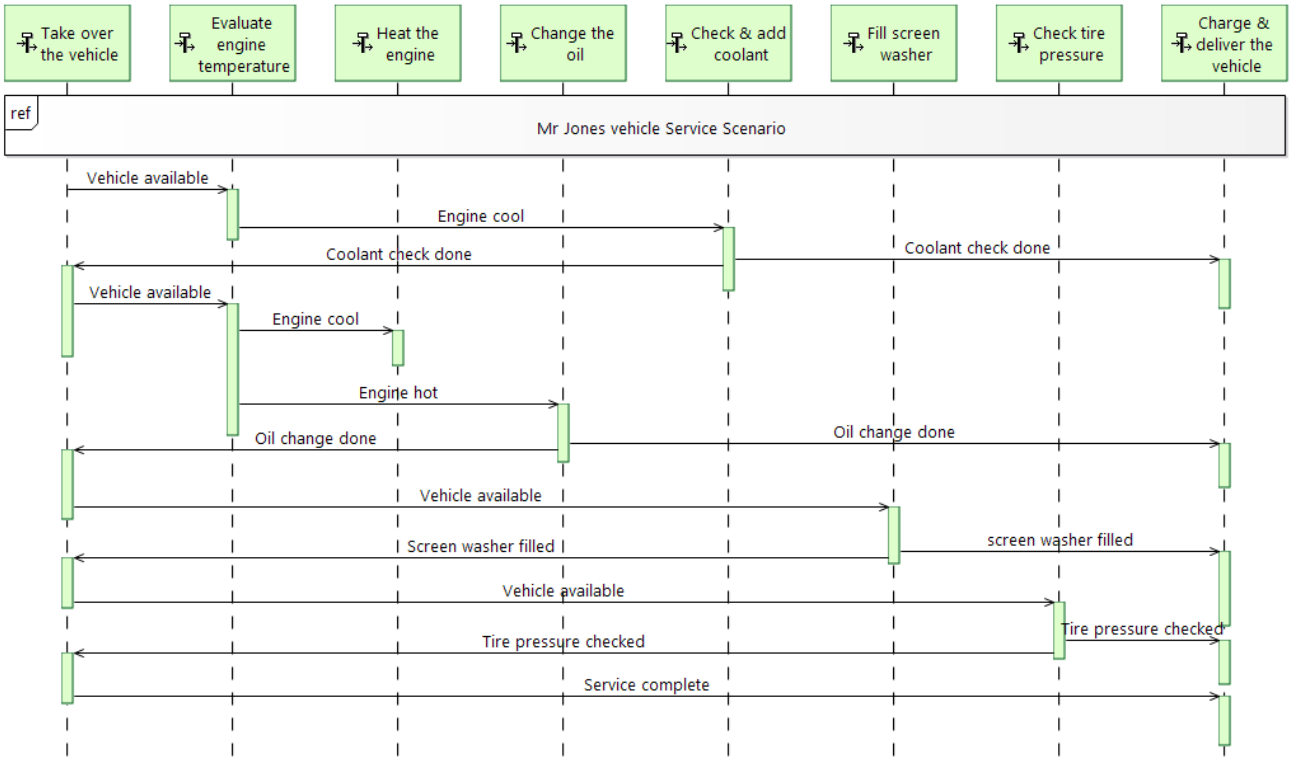

Figure 9: Mr Smith' vehicle checkup with cold engine

Management of types and instances

In Thales at least, a significant cultural difference exists between systems and software engineers. The former are more likely to think first in terms of instances. By default, everything is considered an instance; the concept of type only emerges when replication becomes necessary. Software engineers on the other hand typically start with types and instantiate them later on when modeling deployments.

Issue: How to model instances?

The remainder if this section takes a simple but typical example of the kind of analysis performed in Arcadia. Figure 10 illustrates an extremely simplified redundancy architecture allowing to capture and transmit attitude and heading (A&H) information to the crew of an aircraft.

One of the major objectives of Arcadia models is to constitute a reference for non-functional analyses such as performance, safety, etc. Being able to distinguish each occurrence of architecture elements and to associate different characteristics to each of them is mandatory.

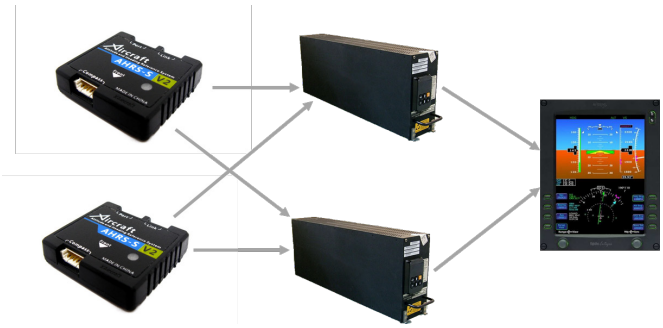

Figure 10: Basic architecture involving redundancy

Conceptually, SysML types, parts, and instances are meant to provide all required constructs to support these kinds of analyses (OMG, 2010). Instead of creating several occurrences of "*AHRS*" and "*Computation Unit*", two types can be used and referenced by parts (Figure 11).

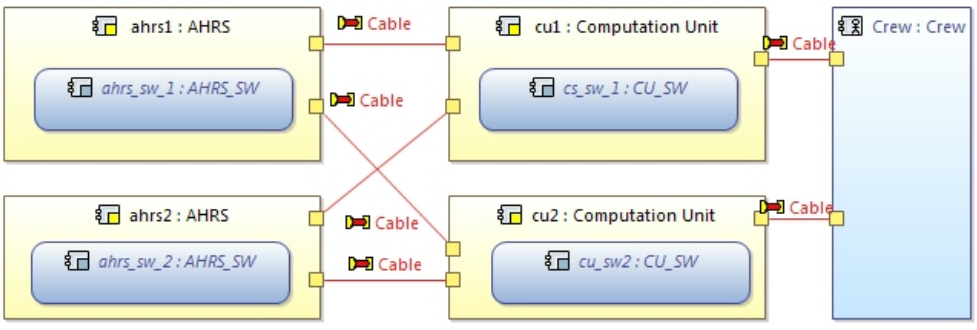

Figure 11: SysML-like Internal Block Diagram

While these mechanisms work reasonably well for structural elements, they do not scale well when functional elements are to be taken into account. Even though SysML actions can be allocated to partitions representing either parts either blocks, the management of elements at

instance level is today considered a weakness of SysML: the FAQ of the SysML Forum[3] states, "Instance Specifications are ambiguously defined and poorly integrated with the rest of SysML.". The multiplication of meta-model concepts (functions, instances of functions, function ports, instances of function ports, functional exchanges, instances of functional exchanges, components, instances of components, component ports, instances of component ports, etc.) is far too complex to most system engineers with no strong UML background.

For the sake of simplicity, functions (instances) in Arcadia are allocated to components (types). Figure 12 shows what the preceding diagram becomes when functions are displayed: a same function is represented by several graphical boxes and a same functional exchange is represented by several graphical links. In this example, the connection between "*Elaborate A&H*" in "*ahrs1*" and "*Choose A&H Source*" in "*cu2*" is a valid one contributing to the redundancy scheme. But the one between "*Choose A&H Source*" in "*cu2*" and "*Compute A&H Graphics*" in "*cu1*" is not.

In Arcadia, functional chains are sets of functions traversals and exchanges, emphasizing specific paths subject to latency constraints, safety expectations etc. Representing a functional chain on such a diagram would be extremely complex, as no distinction can easily be made between the multiple graphical objects representing the same model element.

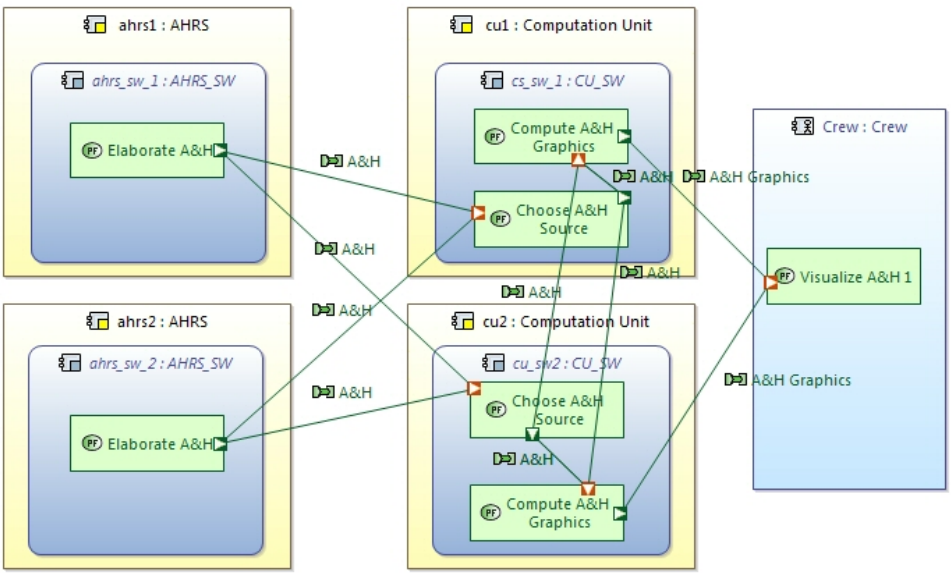

Figure 12: Problems with functions when types and parts are used for components

Figure 13 illustrates a model where the ambiguities of Figure 12 are resolved. Each green box is a unique occurrence of function, each graphically appearing function port and functional exchange is a distinct model element to which specific property values can be given. Links connect occurrences. Functional chains can easily be defined and visualized: each of them traverse given occurrences of each function and functional exchange. This model would be precise enough to support, for example, safety rules checking and failure propagation analysis. For the sake of readability, only two functional chains are displayed on the Figure, but two other equivalent ones can also be defined using "*Elaborate A&H 2*" as an input.

3 SysML Forum: http://sysmlforum.com/sysml-faq/sysml-as-architecture-modeling-language.html

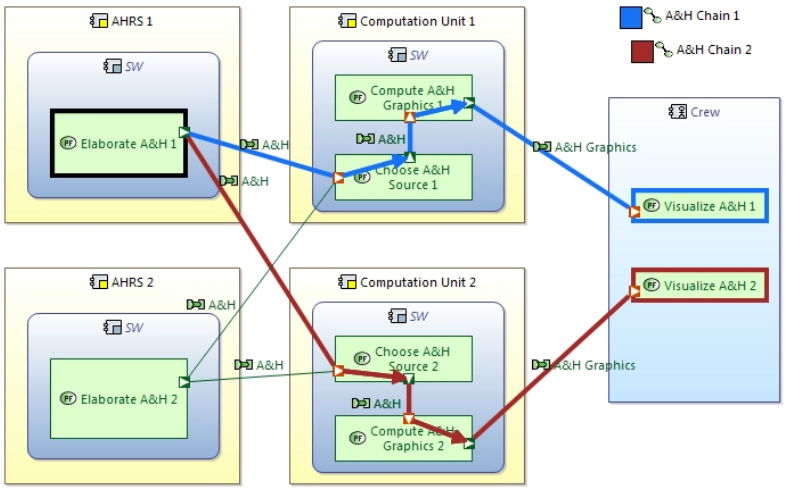

Figure 13: Instance-level architecture diagram

The Capella solution: replicable elements

The concept of part exists but is actually hidden in Capella, as illustrated by Figure 14. Each component is considered as an instance by default. From an implementation point of view, this choice is equivalent of applying the SysML "PropertySpecificType" stereotype on each part. In SysML, the property-specific type implicitly creates a block subclass that types the part in order to add the unique characteristics (Friedenthal et al., 2012).

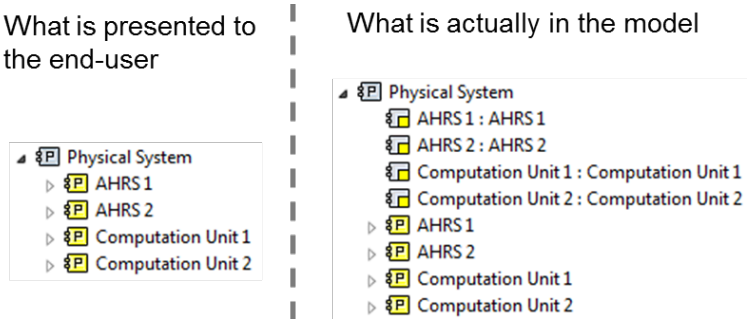

Figure 14: Hidden part concept in Capella

This approach enables simple native instance-level modeling. However, being able to reuse elements and have the equivalent of "types" or "definitions" for these elements is a must-have in any architectural design solution. In Capella, this is implemented with the concepts of Replicable Elements Collections (REC) and Replicas (RPL). An analogy can be made with RECord and RePLay. A REC is the definition of a reusable set of model elements while a RPL is one single usage of a REC in a given context.

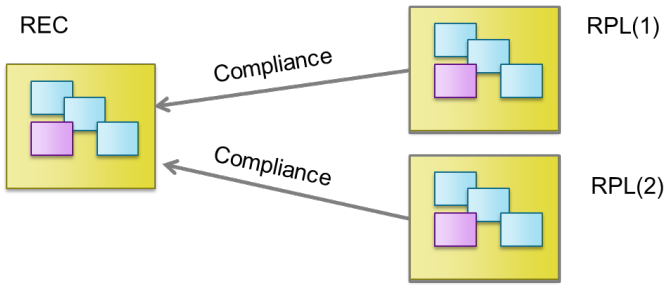

Figure 15: RECs and RPLs

Conceptually, the relationship between a REC and a RPL is close to an instantiation one. A customizable conformity relationship is defined between REC and RPLs, as shown in Figure 15. Black-box conformity means no deviation is tolerated on the RPLs. Constrained-reuse conformity means potential changes are restricted (for example, allocating additional functions to a component without changing its interfaces). Inheritance could be another kind of conformity.

RECs are can be stored in libraries and shared between projects. RECs and RPLs are kept synchronized with dedicated tooling. Figure 16 provides examples of RECs: a REC can be as simple as a single function or component, or as rich as two functions interacting with one another, a functional chain, a group of components, or even an interaction model (sequence diagram). This enables powerful reuse constructs.

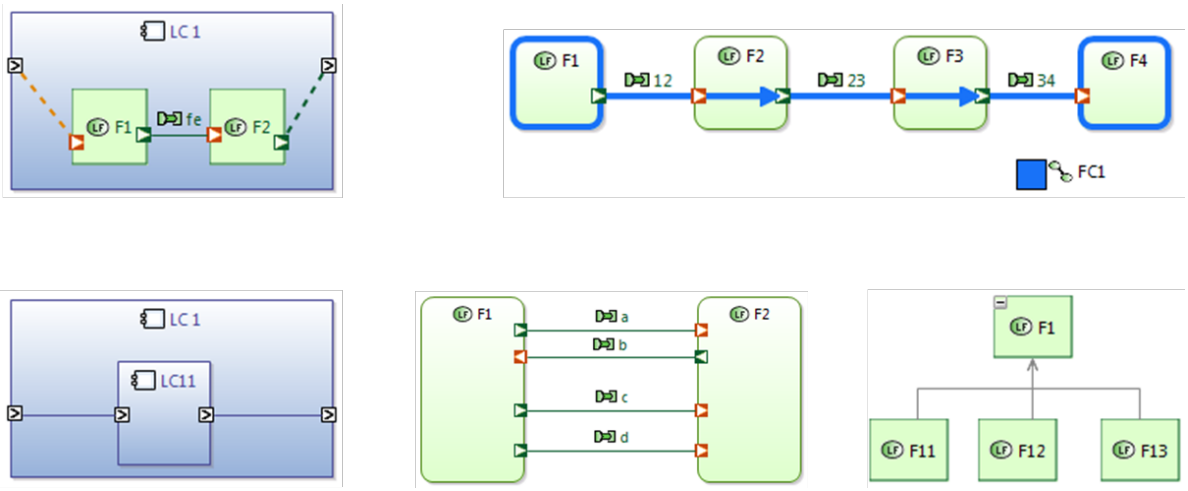

Figure 16: Examples of RECs

Apart from the relationship between a REC and a RPL, the Capella meta-model does not explicitly distinguish types and instances for functions, components, ports, etc. Instead, the nature of a model element is given by its context of usage: a component used to define REC is a type; a component part of a RPL is an instance.

It is interesting to note that a tool like SysML-based Scade Systems[4] also introduced additional reusable/unique blocks mechanisms to overcome the SysML limitations regarding management of instances.

Conclusion

The adequacy of modeling objectives and the means to reach them is critical for the deployment and the adoption of MBSE solutions. After a brief introduction of the Arcadia/Capella solution dedicated to architectural design, this paper elaborates on two concrete problems faced when embracing MBSE approaches on operational projects in Thales.

Functional analysis is a standard practice in systems engineering. There are several ways to implement a functional decomposition, and SysML provides at least two. However, not all are flexible enough to support every situation. The same modeling constructs have to be

4 Scade Systems: http://www.esterel-technologies.com/products/scade-system

compatible with different workflows: abstracting low-level legacy to build synthetic high-level views or refining in the context of an iterative design, among others. Capella provides efficient, straightforward, field-proven means to conduct large functional analyses. Because the focus of Arcadia is architectural design and thus definition of interfaces, this paper explains why the method prohibits the use of certain common constructs like SysML control flows when mixed with dataflow dependencies between functions.

The type/instance paradigm is widely known and easy to understand. However, when confronted with the reality of systems engineering practices and the complexity of the systems to be modeled, the paradigm is difficult to implement. Since Arcadia promotes the evaluation of architectures according to multiple engineering specialties, non-functional viewpoints typically require models at instance level. Managing types and instance on both structural and functional levels does not scale well and SysML shows clear limits. This paper describes the Capella strategy: no real distinction between types and instances at meta-model level and usage of advanced tooling for replication and synchronization of sets of model elements.

The "issues" identified in this paper should not be considered a criticism of SysML. Instead, they are just considered as weaknesses in the context of the Arcadia methodological objectives and of the cultural background of most systems engineers in Thales. The language and diagrams in Capella are actually very close to those of SysML, but are most of the time simplified.

Note that wider aspects like requirements elicitation, concept of operation, system architecture design, modes & states or product line engineering aspects are out of scope of this paper, but they are addressed similarly, based on a DSL approach, focusing on instances and REC/RPL, linked and/or justified against functional analysis. See (Arcadia, 2014) for further details.

Acknowledgements

We would like to thank all the actors of the MBSE community in Thales for their contributions to both Arcadia and Capella. Many thanks as well to Dr Gunnar Schroeter and Pascal Roques for sharing their expertise.

References

Aracic, J. and Roques, P. 2015. "Select and deploy a conceptual modelling language. Some Keys." *Crescendo Technologies*, 5 August.". http://blogs.crescendo-technologies.com

Voirin, J.-L. 2010. "Method and tools to secure and support collaborative architecting of constrained systems" Paper presented at the 27th Congress of the International Council of the Aeronautical Science (ICAS 2010), Nice, France, 19-24 September.

———. 2012. "Modelling languages for Functional Analysis put to the test of real life" Paper presented at 3rd International Conference on Complex Systems Design & Management (CSD&M 2012), 12 December

Sage, A.P. and W.B. Rouse. 1999. *Handbook of Systems Engineering and Management*. Wiley.

Lamm, J. G. and Weilkiens, T. 2010. "Funktionale Architekturen in SysML". In M. Maurer and S.-O. Schulze (eds.), Tag des Systems Engineering 2010, pp. 109–118. Carl Hanser Verlag, München, Germany, November.

Friedenthal, S., Moore A. and Steiner R. 2012. *A Practical Guide to SysML Second Edition: The Systems Modeling Language*. The MK/OMG Press, ISBN-13: 978-0123852069

INCOSE 2014. "A World in Motion - Systems Engineering Vision 2015" Publication

OMG (Object Management Group). 2012. Systems Modeling Language (SysML), Version 1.3

Capella. 2014. "Capella website" http://www.polarsys.org/capella

Arcadia. 2014. "Introduction to Arcadia" http://www.polarsys.org/capella/arcadia.html

Clarity Consortium. 2015. "Clarity website" http://www.clarity-se.org

Biography

Jean-Luc Voirin is Director, Engineering and Modeling, in Thales Defense Missions Systems business unit and Technical Directorate. He holds a MSc & Engineering Degree from ENST Bretagne, France. His fields of interests include architecture, computing and hardware design, algorithmic and software design on real-time image synthesis systems. He has been an architect of real-time and near real-time computing and mission systems on civil and mission aircraft and fighters. He is the principal author of the Arcadia method and an active contributor to the definition of methods and tools. He is involved in coaching activities across all Thales business units, in particular on flagship and critical projects.

Stéphane Bonnet, Thales Corporate Engineering, is the Design Authority of the Thales MBSE workbench for systems, hardware and software architectural design. He holds a PhD on model driven techniques applied to smart cards. From 2008 onwards, he has led the development of Capella. He now dedicates most of his time to training and coaching activities, helping Thales systems engineering managers and systems architects deploy MBSE approaches on operational projects worldwide. He is animating networks of experts from all domains and business units to capture operational needs and orient the method and workbench evolutions and roadmaps.

Véronique Normand is a senior scientist working as a Manager and Design Authority for the Thales Technical Directorate, where she is responsible for the research and technology strategy in model-based engineering for systems and software since 2009. She holds a PhD in Computer Science and a degree from the ENSIMAG informatics and mathematics school in Grenoble, France. Her professional experience involves research in user-centric design, software architecture, collaborative environments, and model-based engineering; consultancy and coaching in engineering methods; software project management.

Daniel Exertier, Thales Corporate Engineering, is Model Driven Engineering Domain Manager, System and Software Technologies Manager. He holds a double MSc degree in Robotics from the Cranfield Institute of Technology (UK) and in Computer Science from the Compiègne University of Technology (France). In charge of the Model Driven Engineering (MDE) domain, he drives the definition of the MDE vision and strategy for the engineering workbenches and builds technical partnerships, collaborative projects and open innovation schemes. In parallel, he manages the MDE Open Source strategy and the relationship with the Eclipse and PolarSys Open Source Foundations and Communities.