Incremental Life Cycle Assurance of Safety-Critical Systems

Julien Delange, Peter Feiler, Ernst Neil

Incremental Life Cycle Assurance of Safety-Critical Systems

Julien Delange, Peter Feiler, Neil Ernst
Carnegie Mellon Software Engineering Institute
4500 Fifth Avenue
Pittsburgh, PA 15213-2612, USA
{jdelange,phf,nernst}@sei.cmu.edu

***Abstract*—Finding problems and optimal designs in the requirements phase is more efficient than later phases. However, over-constraining the solution is also sub-optimal since not all information is necessarily available upfront. 'Build-then-test' approaches which insist on developing first requirements, then architecture, then implementation are not suitable for building systems that must be rapidly fielded and respond to ever-changing demands. Our approach, ALISA, is working on integrating four pillars for incrementally building systems which can be shown to satisfy the relevant requirements. Our four key pillars for assuring requirements satisfaction are requirements specifications, architecture models, verification techniques, and assurance case traceability between the first three. In this paper we introduce our approach, and highlight how we are integrating these pillars using an XText-driven DSL and tool meta-model leveraging existing tools and languages. Our current focus is on understanding exactly which requirements are responsible for the majority of design constraints. Identifying this subset promises to reduce architecture design space exploration and verification overhead, increasing delivery cadence.**

I. Introduction

Safety-critical systems function where an error can be mission- or life-threatening. They are carefully specified and designed according to a rigorous process, usually by different collaborating teams. Through the development process, system stakeholders define their goals, engineers define system requirements from them, and architects design the architecture, breaking the system into several layers and parts. Each part/layer is implemented by potentially different teams, tested separately and then integrated. One major issue of this actual process is the late discovery of errors: 80% of implementation errors are found at system integration but studies have shown that such issues (70% actually, shown in Fig. 1) are likely introduced earlier, when defining the system requirements. Thus, these errors could be discovered and fixed earlier, by improving requirements specification and design.

In this paper, we introduce our approach for improving requirements specification and design, called *Architecture-Led Incremental System Assurance*, ALISA. Our approach is traceable and testable from end to end, that is, from system requirements and stakeholders, down to software and verification outputs. This new method connects requirements to other system artifacts (specifications, models, code, etc.), enabling requirements traceability and validation along the development process. It provides assurance of requirements validation early in the development process, reducing certification cost of safety-critical systems through measurably better requirements and compositional verification evidence.

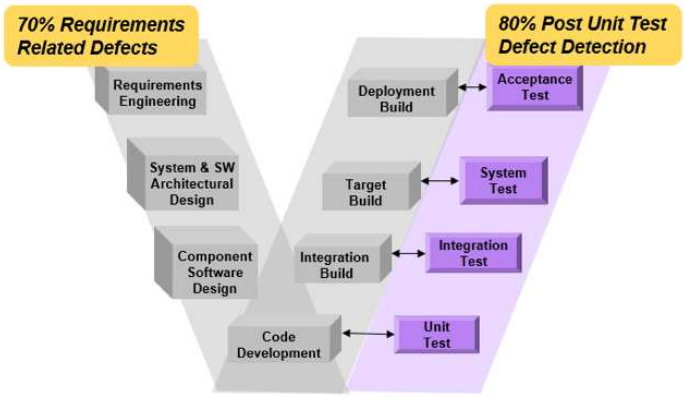

Fig. 1. The Double-V model, showing sources of errors

To improve the quality of requirements we focus on coverage of system specifications, quality attributes, and hazards, as well as management of uncertainty in the requirements. To improve the quality of evidence we use compositional verification, and multi-valued logic to automate the planning, execution of verification plans, and management, reporting of assurance evidence. In order to so we work with three different flavors of incrementality:

1) incrementality by refinement, working with one architecture layer or module at a time
2) incrementality by criticality, focusing on critical requirements/quality attributes first, and then the full set
3) incrementality by change impact, to manage the impact of changes on requirements, architecture design, and verification evidence.

II. Related Work

Moving from requirements to architecture is a key problem in software engineering. Current standards [1], [26] describe the life-cycle process to follow in order to develop software but consistency between development phases is often not synchronized, performed using a manual, labor-intensive process

and consistency between each phase is not automated, and becomes out of date as products evolve.

One important issue has been to identify *architecturally-significant requirements* from a requirements specification; that is, those requirements which will have the most impact on how the system is implemented. For example, a well-developed methodology for this is the Quality Attribute Workshop [5] and design approaches such as the Architecture-Driven Design (ADD) approach. Seminal work includes moving from goals to agent-oriented software, with Tropos [14]; moving from KAOS specifications to software in [27]; and work by Dewayne Perry and his students [8]. One key advance is that there now exist many mature languages for both requirements (e.g., KAOS [9] and the URN standard [15]) and architectures (e.g., AADL [2] and SysML [22]). This allows us to leverage well-known formalisms for the translation. Furthermore, while there were hints in earlier work towards refinement, there was no explicit step for driving evidence-based changes in the requirements, as we propose in ALISA.

In the iterative context, requirements are ideally 'conversation starters' for design elaboration. For example, one takes the provided user story and queries the product owner about any uncertainties. And in an iterative context, particularly with iterations of 2-4 week duration, one can fairly easily refine these requirements. This approach does not work in all contexts, however, and may be guilty of finding local optima (e.g., under-designing), particularly in more complex systems [10].

The most similar approach to ALISA in integrating architecture and requirements is by AutoFocus 3 [4] and Whalen et al. [28]. Whalen et al. describe how SysML can be used with requirements models to accommodate the essential hierarchical nature of system engineering: a flow from more abstract (system requirements) to less (software requirements, then architecture models and code). It is key to recognize that there may be existing architectures and implementations that flow 'upward' to constrain requirements. This paper illustrated that it is often component interaction that provides most system failures, which can be traced to improper requirements decomposition and refinement. AutoFocus 3 [4], from the Fortiss Research Group, is a model-based engineering platform that, like ALISA, supports Eclipse-based end-to-end system development, starting with requirements and finishing with architecture. The two projects have a lot in common; key differences include the languages underpinning the architecture models (AADL in the case of ALISA, a custom language in for AutoFocus), and the verification mechanisms supported (nuSMV model checking in the case of AutoFocus; any AADL compatible verification approach in the ALISA case).

The concept of 'virtual integration' tries to leverage the promise of these model-centric approaches to reduce cost/cycle-time and risk (i.e., rework) by using early, and frequent, virtual integration, illustrated in the System Architecture Virtual Integration (SAVI) initiative ([23], [12]). For simplicity, a related paper [24] suggests doing requirements modeling directly in the architecture-modeling tool (in this case, Simulink). Although adoption and ease-of-use goals are important, our experience suggests this is not ideal for complex requirements models and higher layers of abstraction. For architecture-centric approaches the specific details of the 'architecture' is ambiguous. There are at least four types of architecture we have identified:

- *Functional architectures* capture functional requirements but with little or no information about how those functions will be encapsulated in components.
- *Conceptual architectures* specify how a system is decomposed into software and hardware components and the interfaces between them. Conceptual architectures are used during architecture trade studies and acquisition planning.
- *Design architectures* specify detailed performance characteristics of individual components, including internal design detail to the level required to support the analyses desired.
- *Implementation architectures* specify details needed to integrate and verify an overall system; for example, data that can be used to automatically generate configuration files or perform model-based testing.

This overlapping of abstractions makes it very difficult to properly separate solution context from problem context. Our intent is to provide separate languages (and vocabularies) for discussing these abstractions (for example, requirements specification tools for "functional architectures"), linked with shared identifiers.

In architecture-centric approaches, there is no explicit notation for capturing requirements (as part of a single process). For instance, the values for rate of change of speed thresholds are defined external to the modeling approach. This makes it unclear where and why these values are derived. For example, if the car we are building is a sports car, high acceleration may be desirable. If the car is a minivan (where small children may be more likely), high acceleration may be undesirable. To add traceability and rationale from requirements to architecture, we focus on linking stakeholder goals, system requirements, and an assurance case model for mapping verification strategies to goals. The compositional reasoning from [28] could be integrated as another technique for modeling architecture components and verification strategies.

III. Languages

Defining and verifying requirements rely on several concepts, that are addressed today by separate tools that are not integrated. Each tool covers one or several concepts but does not address the whole process from requirement definition to system validation. As shown in Table I, we distinguish the following concepts, which we call the four pillars of system integration [11]:

- **Requirements and Goal Definitions**: defines the stakeholders, system objectives and requirements. Requirements engineering frameworks offer a formal specification of system requirements, avoiding textual specification. There are several tools to capture and model requirements, such as KAOS [9] or RDAL [7].
- **Architecture Specification**: capture the system architecture structure. This is currently managed by languages such as SysML [22] or AADL [25].

- **Verification**: analyze system artifacts (i.e. model, code, specification), to check requirements. However, these activities are not directly related to the system requirements specifications. This is currently handled by model analysis tools, such as Resolute [13].
- **Claims, Arguments, Assurance**: show how the system enforce the requirements and provides confidence about system quality. This is currently handled by linking verification to requirements and architecture using Structured Assurance Cases (SACM) [21].

One (or several) pillars are supported by existing tools, but, as the sparseness of Table I shows, the artifacts are independent and loosely coupled, and thus, do not cover the entire development process, from requirements specification to system validation. In addition, some tools support the same pillars but with a different (and potentially inconsistent) approach.

For that reason, we propose to unify these concepts. Leveraging existing approaches, we address each pillar with a separate language, and connect them with tool support. Using such an approach, users can then specify their requirements, attach them to the architecture and ultimately, validate them, demonstrating system compliance with the requirements.

We defined the following languages:

- **ReqSpec**: stakeholder goals and system requirements. The language borrows concepts from RDAL [7] and KAOS [9].
- **Verify**: for verification activities, verification plans, methods (i.e. how to analyze and process system artifact to verify a property). This language is based on concepts from SVM [3], JUnit [17] and Resolute [13].
- **Alisa**: for defining assurance work areas, tasks This language borrows concepts from Mylyn [16].
- **Assure**: for assurance case instances. This language reuses concepts from JUnit [17], Resolute [13], SACM [21]

These languages have been implemented within Eclipse using the Xtext [6] framework. Our tool supports requirements/goals specification, validation methods and activities to check requirements enforcement in the architecture/implementation and automatic assurance case generation using the Goal Structuring Notation (GSN) [1] The tools check for requirements coverage (what architecture elements is missing a requirement, and vice versa), consistency (is there conflict between requirements) and auto-generate assurance cases using the GSN notation and tooling from [19] to show how requirements are validated and enforced within the architecture.

Note that we are not committed to a requirements $\rightarrow$ architecture mapping; indeed, in most cases we expect to have an existing architecture and requirements models, so the tracing can be either direction. We call this approach "architecture-led requirements specification" to capture the notion that the architectural model is the central hub of our model-driven engineering approach, with the spokes being the requirements, verification plans, and other artifacts. This does not demand that an architecture exist before requirements, but it does acknowledge that a strictly linear process is not realistic.

[1] http://www.goalstructuringnotation.info

IV. The Lightbulb Example

We show how to use ALISA languages and concepts on a simple system: a light-bulb being powered by a battery. The objective is that the battery has enough capacity to power the bulb.

The architecture of this system is shown in Figure 2 (the AADL textual representation is shown in Listing 1). It consists of two devices: one battery and one bulb. Both components are connected through a power socket. The goal of our example is to show that the battery has enough capacity to power the bulb.

While the graphical representation does not include AADL properties (that specify power capacity and budgets), the textual representation (listing 1) includes such information to capture the power capacity and budget.

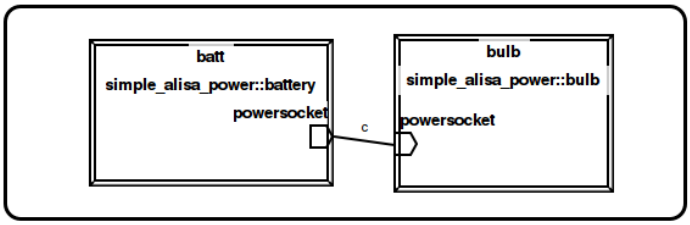

Fig. 2. AADL model of the bulb example

```
package simple_alisa_power

public

with SEI;

bus power
end power;

device bulb
features
   powersocket : requires bus access power;
properties
   SEI::PowerBudget => 60.0 W applies to powersocket;
end bulb;

device battery
features
   powersocket : provides bus access power;
properties
   SEI::PowerCapacity => 80.0 W;
end battery;


system integration
end integration;

system implementation integration.i
subcomponents
   bulb : device bulb;
   batt : device battery;
connections
   c    : bus access batt.powersocket -> bulb.powersocket;
end integration.i;
```

		KAOS	RDAL	Resolute	AGREE	SACM	AADL
Requirements	Stakeholder Goals	X					
	System Requirements	X	X				
Architecture	Specification	X			X		X
	Instance						X
Verification	Activities		X	X	X		
	Verification Methods		X	X	X		
Assurance	Claims and Arguments			X		X	
	Assurance Results			X		X	

TABLE I. TOOL COVERAGE AND GAPS IN SYSTEM PHASES

```
end simple_alisa_power;
```

Listing 1. Architecture of the lightbulb system

To specify the system requirements, the first step is to specify the stakeholders of the system, as shown in listing 2. For this example, we will keep the number of stakeholders to one, the system electrician.

```
organization mycompany
stakeholder electrician
 [ full name "John Doe" ]
```

Listing 2. Stakeholders definition

The next step consists in defining the stakeholders requirements, also known as goals (i.e. what the system is supposed to be and what constraints it is supposed to comply with). Goals definition are written using the **ReqSpec** language, as illustrated in listing 3. A goal is bound to a component (in the present example, the global system), a description and is associated with a stakeholder (the electrician defined before).

```
stakeholder goals mygoals for simple_alisa_power::integration
[ goal g1 : "Power OK" [
    description "We should be able to power the bulb"
    rationale "Without light, we cannot see"
    stakeholder mycompany.electrician
]]
```

Listing 3. Stakeholders Goals (REQSPEC lang.)

Once goals are defined, one should define the system requirements. The **ReqSpec** language is also used to define system requirements, as shown in listing 4. A system requirement is associated with a component, defines a description and is ultimately associated with a goal so that the tool is able to trace goals coverage (what goals are being linked to system requirements) but also architecture validation (what components have requirements).

```
requirement specification myrequirements
for simple_alisa_power::integration
[
   requirement enough_power : "The battery should have \
    enough power"[
       compute actualbudget
      description this "should have a battery with enough \
         power for the bulb"
      see goal mygoals.g1
   ]
]
```

Listing 4. System Requirements (ReqSpec language)

Then, verification plans are defined with the **Verify** language, that specifies how requirements are verified. The assurance plan is separated by claim being validated using analysis tools. Listing 5 shows that the system requirements previously defined are supported by a claim `c1` that checks that the system has enough power. This claim is verified by a verification activity (`mylibrary.electric_requirements`, defined in a general verification library) that will analyze the AADL model and ultimately, check the property values, making sure that the power provided by the battery is more than the power required by the bulb.

```
plan myplan for simple_alisa_power::integration.i [
  claim c1 for myrequirements.enough_power [
    assert all [  mylibrary.electric_requirements
   ] argument "The bulb has enough power"
  ]
]
```

Listing 5. Validation Plan (VERIFY lang.)

Ultimately, the overall assurance plan is defined using the **Alisa** language that defines verification plans are executed (ordering of tests). This is shown in Listing 6, which shows a definition of a basic assurance plan that executes the verification plan defined before (`myplan`).

```
alisa myplan
assurance plan power for simple_alisa_power::integration.i
[ assert myplan ]
```

Listing 6. Assurance Plan Workflow (ALISA lang.)

We have also added a function to automatically export the result of the verification process into an assurance case using the Goal Structuring Notation (GSN) [19]. The objective is to formalize the validation activities into a standardized notation. Our tool export these results into a GSN format that can be processed by the D-case assurance case tool [18].

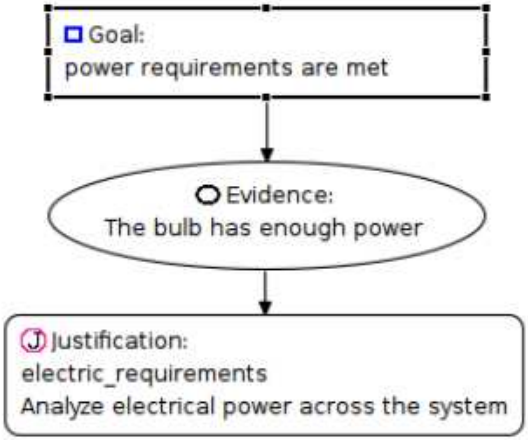

Fig. 3. GSN export example

The GSN diagram of the bulb example is shown in Fig. 3. It details how the goal is decomposed and validated: the stakeholder goal (*Power requirements are met*) is decomposed into an evidence (*The bulb has enough power*) which are ultimately validated in the model (justification *Analyze Power Across The System*). Automating the production of the assurance case would avoid the labor costs associated with the production of such document, but also make them accurate with respect to the actual validation activities done on the model.

V. Larger-scale Example – The System Architecture Virtual Integration Case

In 2009 the SAVI initiative published a white-paper [12] describing a case study outlining how the notion of virtual integration – the use of an annotated architecture model as the single source for architecture analysis – can dramatically reduce rework and verification costs in safety-critical systems development. We have begun to implement that example in the ALISA tool-chain in order to demonstrate ALISA's suitability to realistic problems. In the 2009 report we modeled the sample problem – a Tier 1 airplane system - in AADL, one of the components of the ALISA tool-chain. We have completed this analysis using the ReqSpec tools, reverse engineering the requirements both from the existing architectural documents and pre-existing natural language requirements.

Figure 4 shows the SAVI Tier 1 model components, and figure 5 shows a portion of the Tier 2 model of the Integrated Modular Avionics subsystem. Note that AADL has a robust behavioral specification language, shown in Figure 6.

From these architectural models we focused on the flight guidance subsystem, creating appropriate requirements and verification specifications for these systems. The **Verify** language asserts that a given claim in the requirements (**ReqSpec**) is met. Listing 8 shows the verification plan for the requirements defined in listing 7. The Plugins.ResourceAnalysis verification activity (listing 8) calls a separate, standalone verification tool to check the associated requirement and ultimately returns one of {True, False, Unknown}. The automatic traceability support in our tool propagates the truth-value of each claim to mark requirements as satisfied (claims are True) or unsatisfied (claims are Unknown or False). The result will then be saved and used later to build the associated GSN (as the one shown in figure 3 for the lightbulb example).

We are currently researching better support for more complex propagation of verification outputs. For example, we may use semantics that provide for another verification activity if the first attempt is either Unknown (e.g. a model checker times out) or False (which we are calling fail-then semantics).

```
system requirements ADC_SW : "Requirements for the Software \
Subsystem of the ADC subsystem of the Flight Guidance System"

for Integrator::FGS::ADC::Spec::prAirDataFunction
[
   val UtilRatio = SystemConstants.UtilizationRatio

   val ADC_ProcessingBudget = SystemConstants.TBDm
   val ADC_RAMBudget = SystemConstants.TBDmb
   val ADC_ROMBudget = SystemConstants.TBDmb
   assert ADC_ProcessingBudget <=
      ADC_HW.ADC_ProcessingCapacity*Resource_Utilization_RoT
   assert ADC_RAMBudget <=
      ADC_HW.ADC_RAMCapacity*Resource_Utilization_RoT
   assert ADC_ROMBudget <=
      ADC_HW.ADC_ROMCapacity*Resource_Utilization_RoT

   requirement R1_1: "ADC Processing Budget" [
```

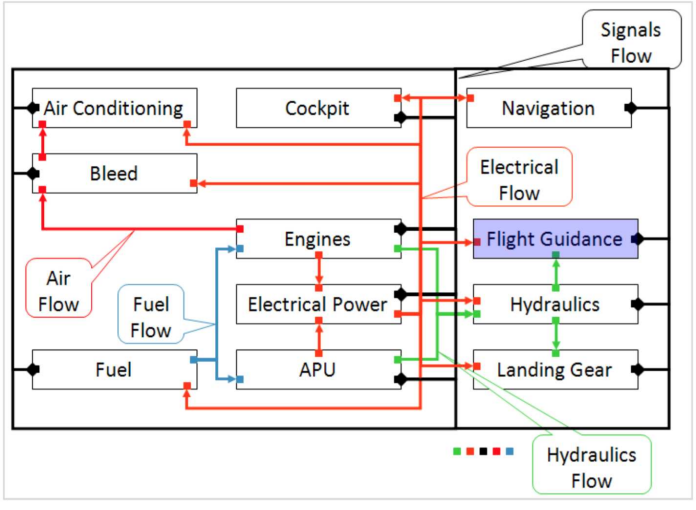

Fig. 4. Tier 1 (system view) of SAVI proof of concept

```
        description "The processing needs of the Software
            Subsystem of the ADC subsystem shall not exceed"
            UtilRatio "percent of"
            ADC_ProcessingBudget
    ]

    requirement R1_2: "ADC RAM Memory Budget" [
        description "The RAM memory needs of the Software
            Subsystem of the ADC subsystem shall not exceed"
            UtilRatio "percent of"
            ADC_RAMBudget
    ]

    requirement R1_3: "ADC ROM Memory Budget" [
        description "The ROM memory needs of the Software
            Subsystem of the ADC subsystem shall not exceed"
            UtilRatio "percent of"
            ADC_ROMBudget
    ]

]
```

Listing 7. FGS Requirements specified in ALISA's ReqSpec

```
verification plan ADC_SWPlan for ADC_SW
[

    claim ADC_SW.R1_1 [
        activities
            processingbudget: Plugins.ResourceAnalysis()
    ]

    claim ADC_SW.R1_2 [
        activities
            RAMbudget: Plugins.ResourceAnalysis()
    ]

    claim ADC_SW.R1_3 [
        activities
            ROMbudget: Plugins.ResourceAnalysis()
    ]

]
```

Listing 8. FGS Requirements specified in ALISA's Verify.

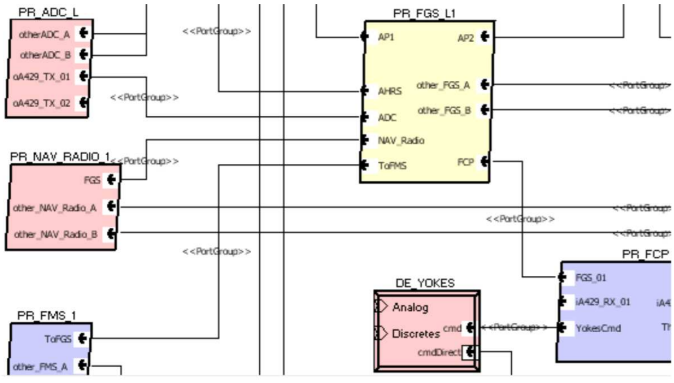

Fig. 5. Tier 2 AADL model of SAVI PoC

```
flows
  fStickToSurface_DirectMode: end to end flow DE_YOKES.fCmdDInput
     -> PortGroupConnection39 -> DE_CONTROL_SURFACES.fSurface
     {
      Latency => 150 Us;
     };
  fStickToSurface_NormalModes: end to end flow DE_YOKES.fCmdInput
     -> PortGroupConnection35 -> PR_FCP.fYokesToFGS1
     -> PortGroupConnection33 -> PR_FGS_L1.fPFCLAWS1
     -> PortGroupConnection23 -> PR_AP_L.fAPCLAWS1
     -> PortGroupConnection10 -> PR_SERVO_AIL.fSurfaceDemandB
     -> PortGroupConnection37 -> DE_CONTROL_SURFACES.fSurface
     {
      Latency => 25 Ms;
     };
```

Fig. 6. Behavioral (flow) modeling in AADL

VI. Conclusions and Future Work

In this paper we presented ALISA, our vision for integrating four pillars for incrementally building systems: requirements specifications, architecture models, verification techniques, and assurance case traceability between the first three. We explained how we created DSL-based tooling to build these pillars, using a simple example. We then presented our example from an avionics domain to support our claim that the ALISA approach promises to reduce architecture design space exploration and verification overhead.

Our current and future work is to work with industry partners to add functional and safety requirements to an existing safety-critical system. We will apply the ALISA tools to perform compositional verification to provide assurance evidence. We are currently translating existing requirements from a DOORS and Excel environment to an ALISA-based requirements and safety hazards specification. We hope to demonstrate measurable improvement in requirements coverage and consistency. This will show the value of ALISA in early project phases. Our ultimate aim is to show measurable reduction in system rework costs by earlier defect detection. The certification process for safety-critical systems is one place where demonstrating compliance can produce large savings, so we are working with a collaborator to to produce additional evidence and certification artifacts to complement their testing evidence. Future work also includes integrating an assessment of requirements uncertainties (such as described by [20]), in order to further circumscribe the set of requirements that need to be checked. For example, in a car we may already be comfortable with our level of knowledge in the anti-lock braking subsystem, but less sure about the new fuel injector. We have also created import and export mechanisms with the OMG's Requirements Interchange Format (ReqIF)[2] in order to facilitate interchange.

Acknowledgments

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of

[2]http://www.omg.org/spec/ReqIF/1.1/

REFERENCES

[1] Iso 15288 systems engineering—system life cycle processes. *International Standards Organisation*, 2002.

[2] S. Aerospace. *Architecture Analysis and Design Language (AADL)*, 2004.

[3] B. Aldrich, A. Fehnker, P. H. Feiler, Z. Han, B. H. Krogh, E. Lim, and S. Sivashankar. Managing verification activities using SVM. In *Formal Methods and Software Engineering*, pages 61–75. Springer, 2004.

[4] V. Aravantinos, S. Voss, S. Teufl, F. Hölzl, and B. Schätz. Autofocus 3: Tooling concepts for seamless, model-based development of embedded systems. In *Model Based Architecting and Construction of Embedded Systems (ACES-MB)*, 2015.

[5] M. R. Barbacci, R. J. Ellison, A. J. Lattanze, J. A. Stafford, C. B. Weinstock, and W. G. Wood. Quality attribute workshops (QAW). Technical Report CMU/SEI-2003-TR-016, Software Engineering Institute, 2003.

[6] L. Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing, 2013.

[7] D. Blouin, E. Senn, and S. Turki. Defining an annex language to the architecture analysis and design language for requirements engineering activities support. In *Model-Driven Requirements Engineering Workshop*, pages 11–20. IEEE, 2011.

[8] M. Brandozzi and D. E. Perry. From goal-oriented requirements to architectural prescriptions: The Preskriptor process. In *International Software Requirements to Architectures Workshop (STRAW)*, Portland OR, May 2003.

[9] A. Dardenne, A. van Lamsweerde, and S. Fickas. Goal-directed requirements acquisition. *Science of Computer Programming*, 20(1-2):1–36, Nov. 2007.

[10] N. Ernst and G. C. Murphy. Case Studies in Just-In-Time Requirements Analysis. In *Empirical Requirements Engineering Workshop at RE*, pages 1–8, Chicago, Sept. 2012.

[11] P. Feiler, J. Goodenough, A. Gurfinkel, C. Weinstock, and L. Wrage. Four pillars for improving the quality of safety-critical software- reliant systems. Technical report, Software Engineering Institute - Carnegie Mellon University, 2013.

[12] P. H. Feiler, J. Hansson, D. de Niz, and L. Wrage. System architecture virtual integration: An industrial case study. Technical Report CMU/SEI-2009-TR-017, Software Engineering Institute - Carnegie Mellon University, November 2009.

[13] A. Gacek, J. Backes, D. Cofer, K. Slind, and M. Whalen. Resolute: an assurance case language for architecture models. In *SIGADA Conference on High integrity language technology*, pages 19–28. ACM, 2014.

[14] P. Giorgini, P. Bresciani, F. Giunchiglia, J. Mylopoulos, and A. Perini. TROPOS: An Agent-Oriented Software Development Methodology. *Autonomous Agents and Multi-Agent Systems*, 8:203–236, 2004.

[15] International Telecommunication Union. User requirements notation (urn). Recommendation Z.151 (10/12), ITU, 2012.

[16] M. Kersten and G. C. Murphy. Mylar: A degree-of-interest model for IDEs. In *International Conference on Aspect-oriented Software Development*, pages 159–168, Chicago, Illinois, 2005.

[17] V. Massol and T. Husted. *JUnit in action*. Manning, 2003.

[18] Y. Matsuno, H. Takamura, and Y. Ishikawa. A dependability case editor with pattern library. In *HASE*, pages 170–171, 2010.

[19] Y. Matsuno and S. Yamamoto. An implementation of GSN community standard. In *International Workshop on Assurance Cases for Software-Intensive Systems*, pages 24–28. IEEE Press, 2013.

[20] A. Nolan, S. Abrahao, P. Clements, and A. Pickard. Managing requirements uncertainty in engine control systems development. In *International Requirements Engineering Conference*, pages 259–264, Trento, August 2011.

[21] Object Management Group. Structured assurance case metamodel (SACM). Specification formal/2013-02-01, Object Management Group, 2013.

[22] OMG. Systems Modeling Language, 2006.

[23] D. Redman, D. Ward, J. Chilenski, and G. Pollari. Virtual integration for improved system design: The AVSI system architecture virtual integration (SAVI) program. In *Analytic Virtual Integration of Cyber-Physical Systems Workshop, 31st IEEE Real-Time Systems Symposium (RTSS 2010)*, 2010.

[24] N. Rungta, O. Tkachuk, S. Person, J. Biatek, M. W. Whalen, J. Castle, and K. Gundy-Burlet. Helping system engineers bridge the peaks. In *TwinPeaks Workshop at ICSE*, Hyderabad, India, 2014.

[25] SAE International. *AS5506 - Architecture Analysis and Design Language (AADL)*, 2012.

[26] R. Singh. International standard iso/iec 12207 software life cycle processes. *Software Process Improvement and Practice*, 2(1):35–50, 1996.

[27] A. van Lamsweerde. From system goals to software architecture. In *International School on Formal Methods for the Design of Computer, Communication and Software Systems: Software Architectures*, pages 25–43, Bertinoro, Italy, September 2003.

[28] M. W. Whalen, A. Gacek, D. Cofer, A. Murugesan, M. P. Heimdahl, and S. Rayadurgam. Your "what" is my "how": Iteration and hierarchy in system design. *IEEE Software*, Mar/Apr:54–60, 2013.