Analysis and Design of Safety-critical, Cyber-Physical Systems

John D. McGregor
School of Computing
Clemson University
Clemson, SC 29632
johnmc@clemson.edu

David P. Gluch
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213 USA
dpg@sei.cmu.edu

Peter H. Feiler
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213 USA
phf@sei.cmu.edu

ABSTRACT

The list of applications classified as safety critical is growing due to emerging contexts such as the Internet of Things that touch the everyday activities of millions of people through smart devices like home automation systems and connected vehicles. These consumer products require high reliability but must be priced competitively. Traditional system development strategies are costly, in part, because traditional verification activities find only a small percentage of defects early in a project and because when discovered late in the development life cycle their repair requires changes to dependent code as well. Our development approach leverages early system architecture knowledge to jump start an architecture-centric development strategy that iteratively establishes traceability among the requirements, architecture, and verification artifacts. A virtual integration strategy makes the current state of the system under development available for analysis early in the product development life cycle. The approach is implemented using the Architecture Analysis and Design Language (AADL) embodied in the Open Source AADL Tool Environment (OSATE). The Architecture-Led Incremental System Assurance (ALISA) toolkit, the latest contribution of our team at the Software Engineering Institute, builds on AADL to provide the constructs and tools for an engineer to specify the integrated system, and to define verification activities that ensure satisfaction of the specification. The results from using the languages and techniques in pilot projects have shown very large cost and time savings, important to holding down costs for consumer-level Internet of Things systems. In this paper we focus on the architecture-led development process and illustrate the support given by ALISA.

1. INTRODUCTION

Smart devices such as fly-by-wire aircraft, connected vehicles, and the "things" connected to the Internet of Things are touching our everyday life and driving the need for more effective and efficient embedded system development techniques. For the Internet of Things to fulfill its potential, devices attached to the Internet must be affordable and safe, a seemingly contradictory set of requirements. In this paper we describe our model-based development process that provides significant development economies to the development of safety-critical embedded systems. These economies are realized through early defect detection while specifying development activities that ensure safe operation of the system under development. We illustrate the technique using a cruise control subsystem for ease of presentation.

The "things" in an Internet of Things system are often cyber-physical systems (CPS) that couple hardware and software to sense the state of the physical world, to then reason about that world and initiate control actions.The use of CPS in essential activities such as control of the living environment and automation of transportation make them critical to the safety of humans. Some of the literature assumes that safety is related to security. That is, connected devices are "safe" as long as our identity is not stolen; however, the use of CPS can pose real physical threats to the health of humans simply due to product defects. Already reported in the literature are defects that threaten the health of the very young and very old through exposure to temperature extremes [3]. Others have hypothesized threats such as elevator doors opening when the elevator car is not present exposing humans to potential life-ending falls [14].

The hardware and software portions of the CPS differ in terms of traditional representations, the degree to which the physical world constrains their operation, and their failure characteristics. These are fundamental issues that must be addressed by a development method for CPS but there are also development challenges that must be addressed by CPS since most are embedded.

Woodward and Mosterman identify five challenges for embedded system development which our development process will face: complexity, optimization, interdependency, verification, and tools [21]. The increasing complexity of embedded systems is often mentioned as one of the system characteristics posing the most immediate threat to embedded system development [15]. Complexity can be addressed by modularity and compositionality and by traceability [1] [18].

Modularity makes inter-dependencies among requirements and architecture components explicit so that they can be developed by distributed teams and managed by tools [17]. Applying modularity operators, such as splitting and substitution given in [2], to the architectural specification results in a hierarchy of component specifications which are interdependent [20]. These specifications can be enforced at every

© 2016 ACM. ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

level in the hierarchy. This hierarchy makes the behavioral requirements and resulting verification obligations explicit. The hierarchy provides a structure that guides the modular definition of the verification activities. Verification activities need to be designed so that they can be applied quickly and maintained easily and the hierarchy of module specifications provides that structure [13]. Tools play a key role in addressing the growing complexity of products, and each of the other challenges [12]. The main focus of this paper is a toolset intended to address the other challenges.

The contribution of this paper is an overview of our model-based development process that addresses these issues and that is supported by a toolset including the Architecture Analysis and Design Language (AADL) and the Architecture-Led Incremental System Assurance (ALISA) toolkit. The process uses assume/guarantee contracts for specifying and verifying safety-critical embedded systems. It builds on existing techniques to detect defects early in the system development process. The rest of the paper is organized as follows: in section 2 we provide information helpful to understanding the development approach; in section 3 we describe the overall development approach; in section 4 we explain the toolset that instruments our approach; in section 5 we give the example development scenario. Finally in section 6 we briefly review our experience in applying the approach and then summarize in section 7 by describing the impact of this approach on the essential qualities of the products created using the process.

2. BACKGROUND

The Architectural Analysis and Design Language (AADL), a standard from SAE, is the foundation of an ecosystem of languages and tools for creating analyzable system architectures [11]. The Software Engineering Institute has developed the Open Source AADL Tool Environment (OSATE), an IDE for designing and analyzing architectures. AADL is strongly typed and provides separation of specification and implementation. AADL provides the ability to define reusable sets of domain specific properties and to define analyses based on those properties.

Members of the AADL ecosystem, e.g. members of the standards committees, commercial users, and research teams, have defined a number of annexes, additions to the core language, that are standardized separate from the core language resulting in a modular family of standards that are easier to manage than a single monolithic standard. Elements in annexes can reference elements in the architecture model coupling the two sufficiently tightly that a change in the architecture is easy to associate with changes to the verification artifacts.

The Error Annex for AADL version 2 (EMV2) is of particular use in designing safety critical systems. The annex focuses on fault propagation, failure behavior of individual components, and composite failure behavior of a system in terms of its components [5]. These three types of specifications allow the designer to supplement component specifications with information about which error types are assumed to be propagated to connected components and which are guaranteed to be contained within the component. This allows the component specification to include nominal behavior and error behavior in similar ways. The error ontology defined in the error annex provides a reference against which the designer can check the component specification to insure that known error types have been included in the specification and mitigated through specific requirements.

AADL is a good modeling choice for CPS since it contains software primitives for processes, threads, and subprograms as well as hardware primitives for processors, memory, buses and devices as well as systems. The extensible property definition facility allows the modeler to accurately reflect those characteristics of the system under analysis that are important to the analyses being conducted. Connections can be defined between the hardware and software elements. Each component can contain nominal and error flows. This allows events (such as interrupts) to be modeled flowing from hardware to software.

Two very powerful property analysis tools, AGREE and Resolute, have been contributed to the AADL ecosystem by Rockwell Collins. Assume Guarantee REasoning Environment (AGREE) is a tool used to define assume/guarantee contracts. AGREE supports compositional reasoning using a Satisfiability-Modulo Theorem (SMT) prover that checks the behavior model for contradictions that would prevent the system from fulfilling the system guarantees. The guarantees are proved by beginning at the lowest levels of each implementation hierarchy, evaluating the guarantees at that level and then composing that evidence at the next higher level of composition to prove the guarantees for the aggregated components.

"Resolute" is a language and tool for developing architectural assurance cases for architectures represented in AADL. Resolute enables the developer to link activities, which create assurance case assets, to the architecture elements being assured. Resolute extends the Goal Structuring Notation (GSN) to provide for defining claims about the properties of the system, identifying the evidence needed to prove a claim to be true, and a structure that sequences the claims and evidence unto an argument. Each claim is a predicate supported by a set of functions to evaluate portions of the architecture description to evaluate the truth value of the predicate.

3. APPROACH

The ALISA project at the Software Engineering Institute is developing a different approach to embedded system design particularly for safety-critical CPS using reusable assets such as reusable requirements, design fragments, and test artifacts. The ALISA toolkit provides a set of domain specific languages to specify requirements, verification activities, and assurance cases. Our approach:

- is based on the hypothesis that even in the very early stages of development the development team has some basic knowledge of, and makes some fundamental assumptions about, the architecture of the system being developed;
- assumes that significant products are built by multiple teams within a large organization or by a consortia of independent organizations that must coordinate their work and communicate technical details with each other; and
- recognizes certain realities of development in this environment such as the need for identifying relationships (1) between development artifacts, such as requirements or reusable components, and the personnel

responsible for maintaining those artifacts, (2) between assets, such as security design patterns, and users of the assets, such as programmers, and (3) between various types of assets, such as requirements descriptions and design patterns that satisfy the requirements.

The development process behind this approach follows the double V model shown in Figure 1 in which the system is constructed in parallel with the specification and evaluation of verification conditions.

Our approach builds on the concept of virtual system integration and analysis of architecture models. Virtual system integration refers to composing a product representation from model components capable of being automatically analyzed. The Aerospace Vehicle Systems Institute (AVSI) chose the SAE AADL standard and tool support as key technology in a proof of concept effort as part of a multi-year System Architecture Virtual Integration (SAVI) initiative. The US Army has recently incorporated this approach under the name Architecture-Centric Virtual Integration Process (ACVIP) in a large scale tech demo program. Our work presented here complements these efforts with a focus on requirements and safety specifications as well as incremental verification throughout the development life-cycle.

AADL is a strongly typed language and supports reliable integration and analysis of subsystem models and analysis of the fully integrated model. The behavior of each model element can be tweaked via properties to more closely express the system under development. The ALISA tools support references to elements in the AADL model.

The virtual integration facet of our approach was the focus of a proof of concept effort [9]. The trials have shown significant cost reductions resulting from the early detection of defects [19]. Our recent work has supported the virtual integration of test assets into effective verification activities. The full development approach is being used on a trial basis in a phased project involving a government agency and multiple contracting organizations.

4. TOOLS

The AADL team at the SEI has developed a set of domain specific languages that support the ALISA approach. The ALISA toolkit combines with other tools in the AADL ecosystem to support the ACVIP style process [6]. In this section we give an overview of three languages, which we will use in a development scenario in section 5. The tools are based on xText, an open source tool for building domain specific languages (DSL) and supporting tools such as context-sensitive editors. Each file and the associated language tools are associated via the file extensions. For example, the stakeholder goals are defined in a file that uses a .goals extension. Opening such a file invokes the appropriate editor.

Reqspec is a requirements specification language that supports the expression of requirements. Each requirement can be linked directly to elements of the AADL model and to other system artifacts. It supports the statement of stockholder's goals (.goals), as shown in Figure 3, and the statement of system requirements (.reqspec), as shown in Figure 5. Requirements are first class objects with properties including links to other requirements and verification activities, stability measures, and ownership information.

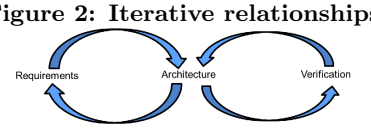
Verify is a language for defining actions to be taken to verify aspects of the system under development. At the most detailed level individual verification activities such as executable test cases or model analyses are specified. A verification suite is a logical grouping of verification activities. Each suite selects the activities needed to achieve a verification objective such as "every line of code" test coverage. A verification plan groups verification suites to meet a project objective such as a type of certification. These verification activities can be mixed and matched in many different configurations to support the verification of different but similar products. Subsystems, and the corresponding verification activities, can also be exported for use in other products.

Assure is a language for defining assurance cases. An assurance case is a structured argument in which claims are made about the degree to which requirements are satisfied by the system development artifacts. A .assure file organize verification results into arguments establishing the validity of the specific claims in the assurance case. Assure supports the automation of the verification plans that execute verification activities.

5. A SCENARIO

In this section we illustrate our architecture-led systems engineering process for developing safety-critical embedded CPS through a scenario. We illustrate the initial iteration which covers some setup and early system development. The system is developed in increments, significant sets of system capabilities. Each increment is developed through a complete requirements / architecture / verification cycle, illustrated in Figure 2. The knowledge gained from making architecture decisions feeds back into the requirements in several ways. Values of properties may be more precisely stated. New requirements, such as safety requirements, may be added. As architecture elements are defined, verification activities are defined and tied to individual requirements and individual architecture elements.

Figure 2: Iterative relationships



We will use a simple adaptive cruise control system as the system under development for our example scenario. The system can be set to maintain a minimum speed and a minimum gap between our vehicle and a vehicle in front. The system has inputs from multiple sensors measuring current speed and size of the gap. The system outputs signals to the Human Machine Interface (HMI) to illuminate icons regarding the state of the system and outputs signals to the throttle and brake actuators. The system is part of a family of cruise control subsystems: cc - traditional cruise control; acc - adaptive cruise control; and cacc - a collaborative, adaptive cruise control.

The system development process begins with the definition of the high level system goals based on stakeholder inputs. The stakeholder goals for a CPS are often stated in terms of quality attributes. These high level goals lead the development team to anticipate that a system will be based on a certain architecture style. The stakeholder goals for the

Figure 1: Parallel development and verification processes

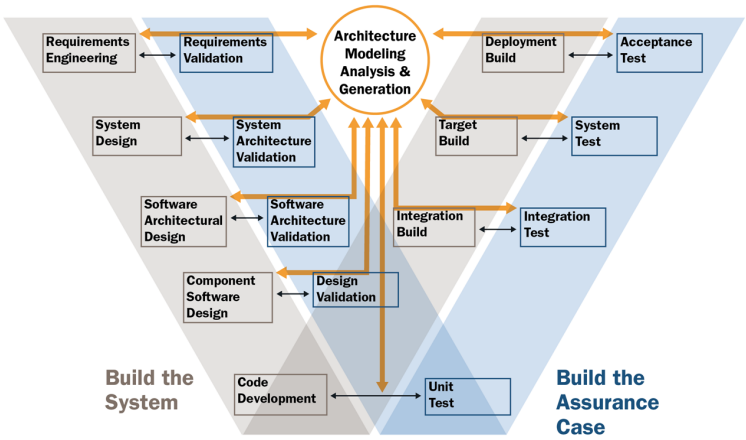


Figure 3: Stakeholder goal

```
1.  stakeholder goals caccStakeholderGoals for caccIntegration::cacc_rt
2.  use constants caccConstants
3.  [
4.
5.    description "Stakeholder goals for the family of cruise controls"
6.          goal g1 : "Safety" [
7.          description "The system shall only change modes when it is safe to do so."
8.          rationale "This is a control system, whose failure affects lives. "
9.          stakeholder caccProject.rs
10.         category quality.safety
11.   ]
```

Figure 4: Feedback/control loop

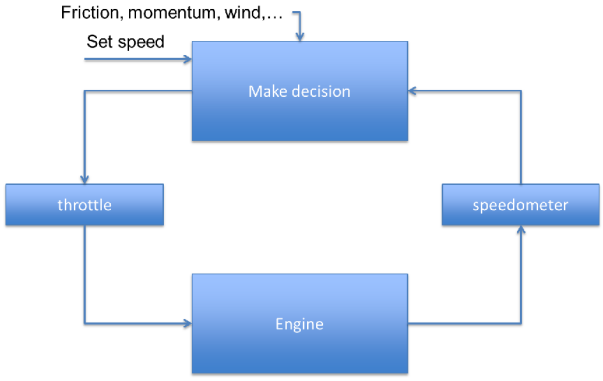


Figure 5: A safety requirement

```
1.  system requirements caccreqs:"CACC"
2.  for caccIntegration::cacc_rt.devices
3.  use constants caccConstants
4.  [
5.    val MaximumSpeed = 120

6.  requirement speed_R1 : "throttle cannot exceed the maximum setting"
7.  [
8.    description this " shall have a maximum reading that is less than or equal to maximum setting"
9.    rationale "fly by wire may introduce an electrical error beyond the physical throttle setting"
10.   compute CurrentSpeed:integer
11.   value predicate CurrentSpeed < MaximumSpeed
12.   mitigates "Invalid data sent by the speedometer"
13.   issues "need to recognize that physical subsystems can present issues for a digital system"
14.   see goal caccStakeholderGoals.g1
15.   category kind.cc kind.acc quality.safety
16.   uncertainty[
17.     volatility 2
18.     impact 3
19.   ]
20. ]
```

cruise control system are to safely and securely constrain the speed of the vehicle.

An example stakeholder goal written in the Reqspec notation is shown in Figure 3. Line 1 shows that all of the goals defined in this file refer to the architecture defined in the cacc_rt specification written in AADL. Line 6 defines Safety as the goal and labels it "g1." The field "stakeholder caccProject.rs" on Line 6 refers to a system stakeholder defined in an organization description using a file with extension .org and shown in Figure 6. This traces the statement of a system goal directly to the stakeholder responsible for the goal.

As the terrain over which the vehicle is traveling changes the system will have to speed up or slow down the vehicle. The team recognizes that these goals point to a standard reference architecture - the feedback control loop architecture style, shown in Figure 4. This initially chosen style drives a high-level AADL model. The stakeholders' goals and the feedback control loop lead to the definition of requirements to assure safe operation.

An example system safety requirement, stated in the Reqspec DSL, for controlling the maximum speed to mitigate one form of unintended acceleration, is shown in Figure 5. This requirement is a safety constraint that enforces a maximum limit in the speeds at which the cruise control can operate. Note that in Line 2 the reference to the AADL model is to a specific implementation, cacc_rt.devices, rather than just to the specification, cacc_rt. One goal of this research effort is to improve traceability from development artifacts back to the relevant requirements. Notice that the system requirement, shown in Figure 5, has a field on Line 14 "see goal caccStakeholderGoals.g1" linking the system requirement to a stakeholder goal. In Figure 5 the fragment

"for caccIntegration::cacc_rt" in the first line links the set of goals to a system specification in the AADL model, shown in Figure 7.

Figure 6: stakeholder description in an organization

```
1. organization caccProject
2. Stakeholder rs [
3.   full name "Roselane Santana Silva"
4.   title "Researcher"
5.   role "System modeler"
6. ]
```

Figure 7: portion of AADL model of adaptive cruise control

```
1.  system cacc_rt extends abstracts::cacc
2.  features
3.  sensed_speed:in data port
    data_types::speed{securityProperties::entryExitPointPrivileges=>5.0;securityProperties::entryExitPointAccessRights=>3.0;};
4.  sensed_speed_limit:in data port
    data_types::speed{securityProperties::entryExitPointPrivileges=>5.0;securityProperties::entryExitPointAccessRights=>3.0;};
5.  sensed_position:in data port
    data_types::gps_Position{securityProperties::entryExitPointPrivileges=>5.0;securityProperties::entryExitPointAccessRights=
    >3.0;};
6.  gap:in data port{securityProperties::entryExitPointPrivileges=>5.0;securityProperties::entryExitPointAccessRights=>3.0;};
7.  gapLimit: in data
    port{securityProperties::entryExitPointPrivileges=>5.0;securityProperties::entryExitPointAccessRights=>3.0;};
8.  power: in data port{securityProperties::entryExitPointPrivileges=>5.0;securityProperties::entryExitPointAccessRights=>3.0;};
9.  throttle_position: in data port
    data_types::cmd{securityProperties::entryExitPointPrivileges=>5.0;securityProperties::entryExitPointAccessRights=>3.0;};
10. end cacc_rt;
```

In Figure 8 the expressions in {} are property specifications. In this case the properties are describing the attack surfaces of the system's subcomponents, a form of security metric [16]. These properties are manipulated by Resolute queries intended to compute the attack surface of the system. One of the Resolute claims is shown in Figure 11. The use of these local property values allows the hardware and software in a CPS to be configured differently for different products within the same product family.

Figure 8: Property Specifications

```
1. Security_Features(self : component) <= **"Calculating total attack surface " **
2. let featureSet : {feature} = features(self);
3. let sumSurfaceC:real = sum({surfaceAreaC(t) for (t:featureSet)});
4. let sumAccessC:real = sum({surfaceAccessC(w) for (w:featureSet)});
5. Req5(sumSurfaceC/sumAccessC)
```

Now that some requirements have been defined, an architecture style identified, and links established among them, the developer turns to verification and defines a set of verification activities using the Verify DSL. A portion of a verification plan is shown in Figure 9. The verification plan is linked to a set of requirements by the "for caccreqs" statement in the header. Each claim uses a specific requirement identifier from the linked set, for example speed_R1 from Figure 5, to link test activities to a specific requirement.

Figure 9: A portion of a verification plan

```
1.  verification plan CACCPlan for caccreqs[
2.  description"This is the verification plan for the requirements in caccReqs"
3.  claim speed_R1:"The vehicle does not exceed maximum speed" [
4.  rationale "This plan achieves a certification level of verification"
5.  activities
6.  speed_ActTest1:"Test of speed control":
7.  caccVerificationMethods.comp(MaximumSpeed,CurrentSpeed)
8.  property values ()
9.  [
10. weight 2
11. timeout 5
12. ]

13. speed_ActTest2:"Second test of speed control" :
14. Plugins.ElectricalPower()
15. property values()[
16. category kind.cc
17. weight 3
18. timeout 5
19.   ]

20. assert all[speed_ActTest1, speed_ActTest2]
21. ]
```

caccVerificationMethods.comp (MaximumSpeed, CurrentSpeed) invokes a user-defined Java method to compute a comparison while Plugins.ElectricalPower(), a calculation of the electric power needs of the system, invokes one of the analysis methods built into OSATE as Eclipse plugins. Other methods may be written in Resolute (see below), as tests in the JUnit test harness, or maybe as manual processes, such as review of a document.

Each verification activity specifies a weight that expresses the importance of the activity as well as a timeout limit to aid in automatic monitoring of test execution. The weight is used in planning which tests to run and evaluating the test results. The category field with "cc" is an enumeration value, which refers to one of three product types introduced earlier in this scenario, used to filter out tests that only apply to the acc or cacc product types.

Finally the developer wants to tie the new increment into the evolving assurance case. The Assure DSL allows the developer to specify the scope of an assurance case. First, the scope of the assurance case is determined by the verification plans of the subsystems identified to be part of the case and those assumed to be verified separately. Second, the developer can specify a filter for subsets of requirements and verification activities to focus on certain functional and non-functional requirements and verification activities relevant to the phase of development. The configured assurance case is then the basis for automated verification execution and tracking of verification results incrementally throughout the project.

Figure 10 shows the contents of a .alisa file that defines an assurance case composed of assurance plans, which in turn are composed of assurance tasks. The assurance case is scoped by the "for" statement to cover the cacc_rt specification while the assurance plan is scoped to the cacc_rt implementation of cacc_rt.devices. This hierarchical definition approach allows for precise targeting of test activities and their reuse.

Before starting the next increment of product definition, the developer can activate the assurance case analysis which will identify problems early. The OSATE environment supports instantiating the architecture using the structures and

Figure 10: A portion of an assurance plan

```
1.  assurance case caccAssuranceCase:"Assurance case for the family of cruise control systems-CACC" for
    caccIntegration::cacc_rt[

2.  assurance plan caccAssurancePlan: "The assurance case for the CACC implementation"
3.  for caccIntegration::cacc_rt.devices [
4.  assure CACCPlan
5.  issues "this assurance plan is related to the assurance case for CACC"
6.  ]

7.  assurance task firstTest:"assurance task for one element" [
8.  category kind.cc kind.acc kind.cacc quality.security quality.safety

9.  ]
10. ]
```

properties defined in the AADL model. This virtually integrated system can be analyzed using a number of analyses that already exist in OSATE and using new queries defined in Resolute. For example, if all the threads in the system are expected to be dispatched periodically the Resolute query shown in Figure 11 would check the entire instantiated model to verify that each thread has the "periodic" property. Line 5 shows the Resolute function that can extract the value for a specific property in the AADL model. This means that many types of defects, mismatched assumptions, and violated constraints can be detected very soon after they are injected into the system. There is also a type of verification activity type labeled "manual" to support activities that cannot be automated but whose results contribute to the assurance case.

Figure 11: A structural Resolute query

```
1. SystemWideReq1() <= ** "All threads have a period" **
2. forall (t: thread). HasPeriod(t)
3.
4. HasPeriod(t : thread) <= ** "Thread " t " has a period" **
5. has_property(t,Timing_Properties::Period)
```

Our approach has integrated safety analysis with requirement specification and verification [8]. It consists of two steps: first, users systematically annotate the architecture with potential error source and propagation specifications. They represent exceptional conditions that have safety implications. The annotated model is then analyzed by verification activities to assess the impact of potential faults (Failure Mode and Effect Analysis (FMEA)) and determined all possible contributors and the probability of occurrence of system failures using Fault Tree Analysis (FTA). In addition verification activities include assuring coverage of potential fault conditions by utilizing a fault taxonomy defined as part of the EMV2 annex to of AADL.

The developer has now completed an initial cycle and is ready to tackle another increment of capabilities. Analyses can be run at any time utilizing the development assets and the rich structure of relationships among those assets. Those relationships are summarized in Figure 12.

6. EVALUATION OF THE APPROACH

The principles and techniques underlying the approach described in this paper have been evaluated in several contexts. They have been used in investigations of representative industrial application systems but not in a production mode [7],[4][10]. It is also being used in a large technical project that does represent production mode.

7. SUMMARY

The five development challenges presented in section 1 provide a basis for summarizing the work related to AADL and ALISA.

- **complexity** - Our approach is modular. AADL provides for defining component specifications in a type-safe manner. AADL has constructs for specifying hardware and software components, and even abstract components, which may be resolved to either. The modeling language supports separating specification from implementation and strong typing of artifacts. Our new DSLs treat many development artifacts as first class citizens about which analysis procedures can reason.

- **optimization** - ALISA supports an iterative style of development. The OSATE environment and add-ins support virtual integration. The AGREE and Resolute annexes support evaluation of predicates to evaluate a number of the properties of the system allowing the design space to be searched and designs to be optimized.

- **interdependency** - The natural dependencies among requirements, the architecture, and the verification conditions are made explicit by the cross-references among the artifacts in the ReqSpec and Verify definitions. The tool set uses those dependencies to locate elements defined in various files for carrying out automated analysis and verification activities.

- **verification** - The Verify and Assure languages provide several tools for defining, automating, and executing verification activities and reporting results. Verification plans are developed for individual modules and for compositions of modules. The activities can be used in multiple assurance plans to develop assurance cases for individual products. Application of these activities is automated in a manner similar to JUnit enhancing reusability.

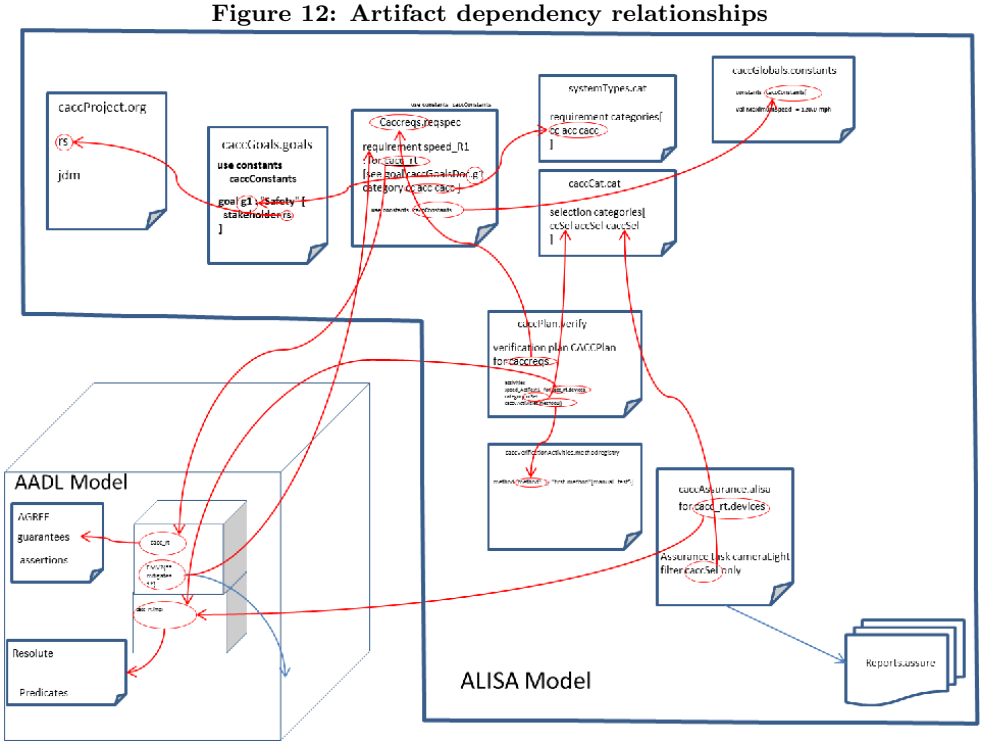
- **tools** - Our approach uses a number of tools beginning with OSATE, the integrated development environment for AADL models, incorporating tools for model checking, such as AGREE and Resolute, developed by others, and a set of domain specific languages aimed at implementing an architecture-led development approach.

We continue to validate our work by modeling and analyzing example systems across a variety of safety-critical domains.The ALISA toolkit is open source and available at https://github.com/osate/alisa.

8. ACKNOWLEDGEMENTS

Roselane S. Silva of the Federal University of Bahia, Brazil, developed the AADL model from which we extracted excerpts. Many thanks to her for her contribution. The full

Figure 12: Artifact dependency relationships



model can be found at https://github.com/rose2s/AADL/tree/master/CACC_model.

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 the Software Engineering Institute, a federally funded research and development center.

DM-0003803

9. REFERENCES

[1] A. Albinet, J.-L. Boulanger, H. Dubois, M.-A. Peraldi-Frati, Y. Sorel, and Q.-D. Van. Model-based methodology for requirements traceability in embedded systems. In *Proceedings of 3rd European Conference on Model Driven Architecture Foundations and Applications, ECMDA'07*, 2007.

[2] C. Y. Baldwin and K. B. Clark. *Design rules: The power of modularity*, volume 1. MIT press, 2000.

[3] N. Bilton. Nest thermostat glitch leaves users in the cold. http://www.nytimes.com/2016/01/14/fashion/nest-thermostat-glitch-battery-dies-software-freeze.html?_r=0. url visited June 29, 2016.

[4] D. De Niz, P. H. Feiler, D. Gluch, and L. Wrage. A virtual upgrade validation method for software reliant systems. Technical Report CMU/SEI-2012-TR-005, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2012.

[5] J. Delange and P. Feiler. Architecture fault modeling with the AADL Error-Model Annex. In *2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 361–368, Aug 2014.

[6] J. Delange, P. Feiler, and N. Ernst. Incremental life cycle assurance of safety-critical systems. In *8th European Congress on Embedded Real Time Software and Systems (ERTSS 2016)*, 2016.

[7] J. Delange, P. Feiler, D. Gluch, and J. Hudak. AADL fault modeling and analysis within an ARP4761 safety assessment. Technical Report CMU/SEI-2014-TR-020, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2014. url visited June 29, 2016.

[8] P. Feiler, D. Gluch, and J. D. Mcgregor. An architecture-led safety analysis method. In *Proceedings of ERTS 2016*, 2016.

[9] P. 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, Pittsburgh, PA, 2009.

[10] P. Feiler, C. Weinstock, J. Goodenough, J. Delange, A. Klein, and N. Ernst. Architecture-led diagnosis and verification of a stepper motor controller. In *8th European Congress on Embedded Real Time Software and Systems (ERTSS 2016)*, 2016.

[11] P. H. Feiler and D. P. Gluch. *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*. Addison-Wesley Professional, 1st edition, 2012.

[12] P. H. Feiler, B. A. Lewis, and S. Vestal. The sae architecture analysis: Design language (aadl) a

standard for engineering performance critical systems. In *2006 IEEE Conference on Computer Aided Control System Design, 2006 IEEE International Conference on Control Applications, 2006 IEEE International Symposium on Intelligent Control*, pages 1206–1211, Oct 2006.

[13] D. D. Gajski, S. Abdi, A. Gerstlauer, and G. Schirner. *Embedded system design: modeling, synthesis and verification*. Springer Science & Business Media, 2009.

[14] J. Holler, V. Tsiatsis, C. Mulligan, S. Avesand, S. Karnouskos, and D. Boyle. *From Machine-to-Machine to the Internet of Things: Introduction to a New Age of Intelligence*. Elsevier Academic Press, 2014.

[15] P. Liggesmeyer and M. Trapp. Trends in embedded software engineering. *IEEE Software*, 26(3):19–25, May 2009.

[16] P. K. Manadhata and J. M. Wing. An attack surface metric. *IEEE Transactions on Software Engineering*, 37(3):371–386, 2011.

[17] B. Nuseibeh. Weaving together requirements and architectures. *Computer*, 34:115–117, 2001.

[18] K. J. Sullivan. The structure and value of modularity in software design. In *SIGSOFT Software Engineering Notes*, pages 99–108. ACM Press, 2001.

[19] D. Ward. Avsi's system architecture virtual integration program:proof of concept demonstrations, 2013.

[20] M. Whalen, A. Gacek, D. Cofer, A. Murugesan, M. Heimdahl, and S. Rayadurgam. Your 'what' is my 'how': iteration and hierarchy in system design. *IEEE Software*, 30(2):54–60, 2013.

[21] M. V. Woodward and P. Mosterman. Challenges for embedded software development. In *Proceedings of the 50th International Midwest Symposium on Circuits and Systems (MWSCAS), Montreal, Canada*, pages 630–633, 2007.