Metrics for Integrated Modular Avionics Architecture

Daniel Jankovic

LUND
UNIVERSITY

Department of Automatic Control

MSc Thesis
TFRT-6036
ISSN 0280-5316

Department of Automatic Control
Lund University
Box 118
SE-221 00 LUND
Sweden

Abstract

Integrated modular avionics (IMA) architecture is an emerging concept in the military aerospace industry after being successfully implemented in the commercial domain. The highly modular architecture allows multiple aviation applications to execute on the same hardware thanks to defined standards by Aeronautical Radio, Incorporated (ARINC). System architects are responsible for designing and taking advantage of the IMA architecture to meet the requirements set by the stakeholders. They rely much on experience, system knowledge, and design patterns in their work.

This thesis aims to find relevant metrics for system architects when developing IMA architecture in the aerospace industry. A metric survey, with focus on the aerospace and closely related industries, is conducted and broadened to include software and real-time metrics. To find one to three metrics multiple presentations and screenings are held together with a team of domain experts.

Three metrics are selected: structural complexity using Shannon's entropy, instability and abstractness metric, and complexity and coupling metric. The metrics are described in detail and implemented. A small scale system is created to assist and provide better understanding how and what the metrics are measuring. Whether the selected metrics are employable for system architects in aerospace industry still remains to be empirically validated. A proposed validation process is presented for future work.

Acknowledgements

Thanks to my supervisors Joacim Kämpe, Pär Hammarström at SAAB Aeronautics in Linköping and Anton Cervin at Department of Automatic Control, Lund University for their support, feedback, and guidance throughout the project.

I would also like to take the opportunity to thank my family and one special person in my life for supporting me, not only throughout this project but during my entire period of studies.

Abbreviations

AFDX Avionics full duplex

APEX Application executive

API Application programming interface

ARINC Aeronautical radio, incorporated

AUTOSAR Automotive open system architecture

COTS Commercial-off-the-shelf

ECU Electronic control unit

DFBW Digital fly-by-wire

IMA Integrated modular avionics

LRU Line replaceable unit

MOTS Military off-the-shelf

MLU Mid-life upgraded

OEM Original equipment manufacturer

PCM Package coupling measurement

RTOS Real-time operating system

SAP Stable abstractness principle

SLOC Source line of code

Contents

1

Introduction

1.1 Motivation

Some of the first aviation instruments to be created were analogue navigational aids; altimeter, attitude indicator, and radio navigation [Helfrick, 2007]. The advancements in the electronics domain allowed new technology to be integrated into the aircraft. Federated architecture followed after the older distributed analogue and digital architecture. It was developed in the late 1980s, early 1990, and variants of the architecture is represented in military aircraft operating today e.g. F-16 mid-life upgrade (MLU), SAAB Gripen and Boeing AH-64 C/D [Moir and Seabridge, 2006].

The concept behind federated architecture is ”one function — one computer” principle [Bieber et al., 2012]. The computers, called line replaceable unit (LRU), are located on brackets inside the aircraft and connected through standardised data buses. The goal is to have the possibility of system reconfiguration while reducing maintenance and spare parts cost.

In 1995, Boeing, in cooperation with Honeywell, presented the first integrated modular avionics (IMA) architecture for cockpit functions for the newest 777 aircraft [Pelton and Scarbrough, 1997; Aleksa and Carter, 1997]. The new highly modular architecture introduced the ability to have multiple aviation functions execute on the same hardware. Instead of installing multiple physical computers, one computer could mimic multiple computers through partitioning. Each partition worked as a virtual computer that could execute aviation applications. The benefits of IMA spread to other developers and domains. Airbus, together with Thales and Diehl, proceeded to develop ”open IMA” technology for the A380 programme [Butz, 2010]. The automotive industry began to research and develop their own counterpart. Automotive open system architecture (AUTOSAR) was the result of a joint initiative by multiple companies [Heinecke et al., 2004].

In the aerospace industry, architects and stakeholders work together to achieve the requirements set from each domain e.g. software, mechanical, electrical, and control engineering, within the financial budget. Designing, selecting, and maintaining a system that is highly modular and upgradable is difficult, especially without any metrics to verify the work. In the past system architects have relied on past experience, system knowledge, and design patterns. Metrics can highlight areas of concern and show how the product is developing over time. They can also provide useful information that can be helpful when deciding a design. Having system attributes such as modularity and upgradeability can increase the market value since the product can be tailored to fit customer requirements with a minimum of redesign.

1.2 Challenges

Commercial aviation was early to adopt open architectures, which meant that suppliers could provide commercial off-the shelf (COTS) products to a system that was standardised. The military domain used proven technology that could work in the harsh environment they are forced to withstand. While both the military and commercial aviation developed separate components, it soon became clear that military industry could not keep development at the same speed as the commercial industry which had IT and telecommunications as major driving force for technology achievements [Moir and Seabridge, 2006]. Commercial computing was early to adapt open architectures, whereas military followed along much later.

With IMA architecture there is potential to have a system that is reliable, interchangeable, extensible, and modular while keeping costs down. MLUs and updates can be performed more efficiently without grounding the aircraft for a longer period. The possibility of having aviation applications management similar to a smartphone is an interesting aspect since it can reduce the integration and revalidation time. Creating a system architecture with this in mind is very important since a product's life-cycle depends on it.

System architects are responsible for breaking down complex systems into manageable components that engineers from other domains can handle. System architects are in close cooperation with multiple stakeholders within an organisation to understand different level of requirements, technologies, and development efforts.

System architects in aerospace industry have throughout the years been relying on knowledge, design patterns and past experience when designing new system architectures. For the system architects to agree on a specific architecture, metrics are important to support the decisions. It should be possible to provide insight of the system during multiple phases of a product's life cycle. The usage of metrics in system architecting is rare, and one of the biggest challenges is to define relevant metrics that will help the system architects and stakeholders reach their goal.

1.3 Goals and contribution

The thesis aims to evaluate and present metrics that will be beneficial for system architects and stakeholders when developing new system architectures. The metrics

should assist in decision making, be able to track the development of the product, and highlight areas of concern. The metrics should be applicable during different phases of a product's life-cycle. The goal is to provide a detailed description of how to use and implement the metrics. It also explains how the measurements are generated and visualised. Since time is a factor, suggestions on how the validation of the metrics over time is performed will be provided to verify whether the metrics are useful or not. These findings will hopefully contribute system architects in their work of development when satisfying requirements with the best design choices.
To sum up, the thesis is aiming to provide answers for the following questions:

Q What metrics exist today within system architecting and how are they used?

Q What metrics are relevant to apply when developing an airborne based IMA architecture that should have characteristics such as interchangeability, extensibility, and modularity?

Q How should the metrics be validated against the predefined requirements over time?

1.4 Methodology

Research and implementation

The thesis was divided into several phases, which used different methodologies. The phases are explained in the order of execution.

1. *Survey of metrics within area of interest.* To understand the extent of metrics that are used in system architecture within aerospace industry and surrounding industries, the phase was allocated the majority of time. The research also included real-time and software metrics due to the limited amount within system architecture.

2. *Production of a long-list.* Metrics that were of interest were included into a long-list. See *Metric selection* for further requirements for selection.

3. *Production of a short-list.* Once the long-list was completed every metric was briefly studied. Metrics from the long-list were evaluated with a team of domain experts. The metrics that were considered to be useful were included in the short-list. See *Metric selection* for further requirements for selection.

4. *Selection of 1-3 metrics for implementation.* After an in-depth study of the short-list metrics, the metrics were presented for the same team of domain experts, however, this time with a better understanding how the metrics are used and interpreted. Selecting one to three metrics for implementation was the goal in order to proceed to next phase.

5. *Implementation of the metrics*. The metrics were implemented using high level, but limited, programming language to illustrate how they can be used.

Metric selection

The selected metrics should be applicable throughout a product's life-cycle in order to give system architects and stakeholders guidance and a basis for decisions. The primary goal was to find metrics that are being used by system architects in aerospace industry but also related industries e.g. automotive, rail, space, nuclear, and etc., however, the research could be broadened to include software and real-time metrics depending on the outcome. A team of domain experts, consisting of a system architects with different background and experience levels from aerospace industry, were part of the evaluation process. Their opinions weighed heavily when selecting the presented metrics.

Metric implementation

The selected metrics in the last phase of research are implemented in Matlab. The implementation should verify that the mathematics behind the metrics are correct and provide helpful guidance on how to use them. It should also provide the reader information what input values are needed and how the result should be interpreted.

1.5 Outline

The thesis is outlined in the following order. Chapter 2 presents avionics history and system architecture with the intention of leading up to IMA architecture. Chapter 3 presents the selection process of the research stage and what metrics that were selected. A more thorough description of the metrics is provided. Chapter 4 presents the implementation of the metrics with a simple system model providing understandability. A basic visualisation is presented as well together with how the metrics have been validated theoretically and empirically earlier. It suggests how the metrics should be validated over time for use in systems architecting. The conclusion and future work is presented in Chapter 5.

2

Background

2.1 Avionics

Avionics (from ”aviation electronics equipment”) combines two fields of science that made much advancement in the late 20th century. Among the first aviation instruments to be created were navigational aids; altimeter, attitude indicator, and radio navigation. In the 1920s airliners depended on landmarks and good weather conditions throughout the whole route to perform the flight. Since this also was an issue for the US army, an investigation was started and came to be known as the ”Blind flight” project. During the project there were attempts to install electronics, but as Helfrick explains it, ”Electronics of that period were large and heavy, and the cost for every gram of mass in an aircraft is high.[...]This implies more fuel and higher rates of consumption. Electrical power requirements rise, calling for larger generator and battery, hence more weight and larger engine” [Helfrick, 2007].

The post WWII era saw an evolution in the field of electronics. Semi-conductors proved to have same functionality as the vacuum tubes which introduced the integrated circuits (IC) and microprocessors. These discoveries are considered to have an important role in aviation electronics since the trade-off between size and computational power for aviation function found common ground. With the transition from analogue to digital electronics, the amount of ICs increased in aircraft. System architectural concepts played an important role in organising the hardware and software.

1972 marks the first time in history an aircraft was completely dependent upon an electrical flight control system (FCS). The project, that was founded by NASA, used an A-8 Crusader as test bed for development of the new technology called digital fly-by-wire (DFBW) [Tomayko, 2000]. Instead of having flight controls directly connected to the hydraulics or through an oscillation dampening computer such as the ”SPAK” in the Viggen [*Flight Manual A/C JA37*], the inputs from the controls were converted into electrical signals which the on board digital computers used to calculate the movement of the actuators. For this project three computers were used to calculate the signal from the pilot inputs separately. The output signal to the actuators was based on the outcome of a vote performed by the computers. Apart from reassuring that the system made the right decision it also added redundancy to the system in case one computer crashed.

While the technology advancements were made in the field of electronics more aviation system were implemented on IC. During the 1970s the transition from analogue to digital computers took place. New high-speed buses such as ARINC 429, MIL-STD-1553, and ARINC 629 were being implemented in a more organised architecture in the mid 1980s [Moir and Seabridge, 2006].

From 1988 until the roll out of the first Boeing 777, Honeywell developed, in cooperation with manufacturers, the first generation airplane information management system (AIMS) which used the IMA architecture [Morgan, 2007]. Only a set of system functions were implemented into the AIMS, e.g. primary display functions, flight management function, airplane condition monitoring function, etc. Boeing's analysis of the AIMS on IMA architecture compared to the previous LRU-based approach showed ”lower recurring costs, lower weight, lower volume, less wiring, and less power consumption for the IMA architecture” [Pelton and Scarbrough, 1997].

The European manufacturer Airbus took it one step further and developed ”Open IMA” together with Thales and Diehl for the A380 programme. Airbus selected the open ARINC 600 standard over Honeywell for the avionics modules, replaced the bus with a commercial standard 100Mbit Avionics Full DupleX (AFDX) switched Ethernet network, and applied IMA modules to all types of aircraft functions. By creating a market for the open IMA standard costs could be lowered through competition [Butz, 2010].

The IMA concept is applied in the current generations of commercial aircraft e.g. Airbus A380, A350, and the Boeing 787. The military industry focused on creating military off-the-shelf (MOTS) products that the system designers could apply in the design. The idea was to have the commercial industry support the military products and provide technology that was needed. However, the commercial industry was moving ahead at a greater pace than the military could handle, thus, forcing the military industry to consider open IMA approach [Moir and Seabridge, 2006].

2.2 Avionics architecture

Since 1960s the avionics architecture has had an important role. This section will describe how the architectures have evolved and where we are today.

Distributed analogue architectures

The distributed analogue avionics architecture, Figure 2.1, is implemented in aircraft designed and manufactured throughout the 1950s and 1960s. The aircraft system is composed of interconnected subsystems that are hard wired, making it difficult to modify or expand. Different types of functions are provided through the wiring e.g. power supply, sensor signal voltage, and status signal.

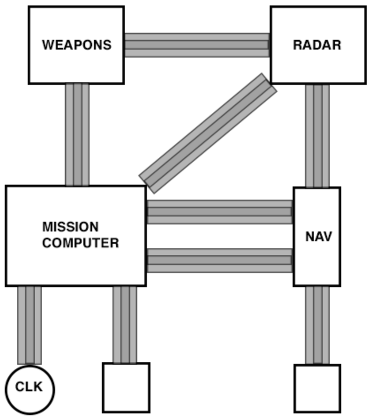


Figure 2.1 Distributed analogue architecture.

Analogue systems are prone to drift due to issues such as electromechanical limitations and temperature variations. Together with the size and performance the system was considered unreliable in that it did not provide accuracy nor stability. Maintaining and servicing required skill, and while spare parts where decreasing the cost of repair only increased. There are aircraft manufactured during this era that are still flying. They have been retrofitted with modern systems to comply with regulations. However, the number of airworthy aircraft is decreasing due to the maintenance costs and structural problems.

Distributed digital architectures

The advancements in digital computing during 1960s soon found its potential as replacement for the analogue equivalent. Far from the size and computational power of today's computers the distributed digital architecture, Figure 2.2, remained unchanged when introduced in 1970s. Computations could now be performed at a higher speed and with greater accuracy which resulted in constant performance and elimination of the drift issues that were in the analogue systems.

During this period the military industry was developing systems along side the commercial aviation industry. The new bus technology between subsystems reduced the weight and cost but it was still complicated to add new functions after the aircraft left the factory. The two half-duplex data buses that were being adopted was the civil ARINC 429 and the military Tornado serial. The popular ARINC 429 became the standard in the commercial industry which paved the way for future ARINC standards in the industry. Boeing 737, 757, 767, Airbus A300, 320, 330, and including some business jets used a simplified version of this architecture with ARINC 429. Jaguar, Tornado, and Sea Harrier are military aircraft that used the architecture with

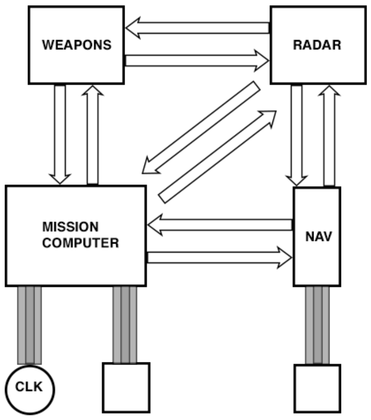
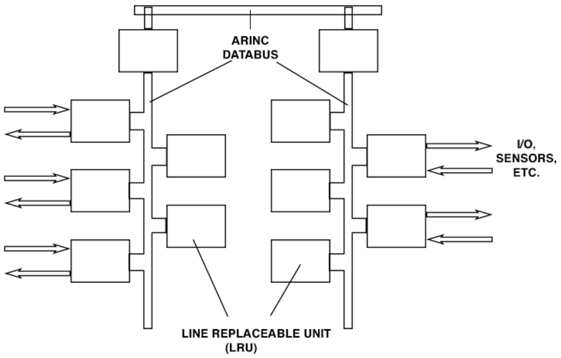


Figure 2.2 Distributed digital architecture.



2

Figure 2.3 Civil federated digital architecture.

the Tornado serial.

Federated digital architectures

During 1980s and early 1990s the civil federated digital architecture, Figure 2.3, and the military recognise that avionic applications share similar attributes and have dependencies across the system. Applications that are related by the same domain are grouped together. The domain can be interpreted as a LRU where each system is an unit with dedicated task-oriented embedded processor with memory. Systems that share data within the domain are connected via local data bus. In turn the domains, or LRUs, share data between each other on higher level using separate data buses.

The military industry developed and adopted the new MIL-STD-1553B data bus and made it available for all members of the North Atlantic Treaty Organisation (NATO). The civil industry was rather late to adapt the federated concept because it had invested in the reliable ARINC 429. It was well established for the purpose it served. While the two different organisations did not agree on implementations and protocols, a new civil standard emerged which came to be known as the ARINC 629. It had higher bandwidth and support for multiple redundant operations for safety critical systems, but it was only used in Boeing 777.

The new bus technology reduced the wiring in the aircraft. This lowered the weight of the aircraft and cost when performing upgrades since no new wiring was necessary. Further advancements in electronics field added support for software re-programming of systems in LRUs. Maintainability increased drastically and upgrades could be performed over night.

Integrated modular avionics

The federated architecture has distributed computing where each application is embedded to a task-oriented computer. It has separate processing, internal buses, point-to-point wiring between the computers and sensors/effector. The logic of the system is represented physically in the architecture. IMA architecture replaces much of the physical hardware in federated architecture with virtual hardware. Figure 2.4 shows an example how an IMA architecture can be designed. Instead of having distributed computing, IMA creates virtual systems using general-purpose processors with shared resource computing. Data exchange between applications is provided by an upgraded network based COTS Ethernet technology that supports real-time features for safety-critical applications. The AFDX is specified in ARINC 664. Apart from providing full duplex AFDX also includes redundancy, determinism, and high speed performance.

Virtual systems demand robust partitioning of the shared resources. Under no circumstances may an application.

- impact the resource when another partition is processing it.
- access the memory belonging to another partition.
- affect the I/O resources of another partition.

It is the task of the real-time operating system (RTOS) to avoid these mishaps. The necessary features are defined in ARINC 653 [Prisaznuk, 2015]. The application executive (APEX) is an application programming interface (API) defined in ARINC 653 that establishes one of the most important attributes, separation between software and hardware in the system.

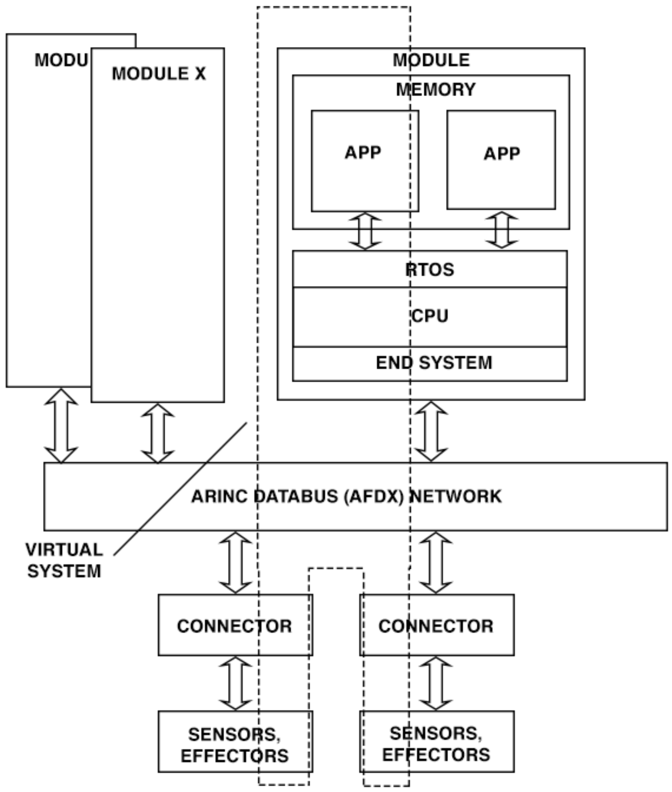


Figure 2.4 IMA Architecture.

3

Metrics

3.1 Survey of metrics

Aboutaleb and Monsuez mention that development of system architecture is unique for every project and therefore conducting an exhaustive metrics survey is hard [Aboutaleb and Monsuez, 2016]. System architects tend to rely on design patterns, experience and intuition but IMA architecture development has a high influence from software development. The lack of metrics from the field of system architecture presented an option to widen the scope of the survey and investigate if metrics for system quality attributes could be applied.

System quality metrics that measure *modularity*, *abstraction*, and *maintainability* among others were used as search key words. Books on design of software architecture, system engineering, and metrics in software engineering were examined to investigate the usage of metrics in different stages of development [Cervantes and Kazman, 2016; Bass et al., 2012; Kan, 1995]. Fundamental computer program and system design metrics were examined [Yourdon and Constantine, 1979]. Because IMA is relying on hard RTOS, real-time metrics were also investigated in order to perform real-time analysis as early as possible in development. Every potential metric that was of interest were collected in a long-list, Appendix A. Below are the metrics included in the long-list. A short description for each metric provides a small insight into what they measure.

- Coupling and cohesion [Yourdon and Constantine, 1979]

  Describes the concept of inter respective intra relationship between components.

- Package coupling measurement [Gupta and Chhabra, 2009]

  Measures coupling between components within a software package.

- Semantic decoupling [Navarro et al., 2010]

  Measures the coupling of mappings between specifications and design principles.

- McCabe's cyclomatic complexity [McCabe, 1976; Kan, 1995]

  Measures complexity by distinguishing the number of linearly independent paths.

- Halstead's complexity [Qutaish and Abran, 2005]

  Measures the complexity of a software using properties from the source code.

- Complexity (Fan-in $\times$ Fan-out)^2 [Henry and Kafura, 1981]

  Measures the complexity in regards to how many inputs a procedure needs respective how many outputs goes out.

- Software design complexity [Card and Agresti, 1988]

  Measures the design complexity through functions that measure the local and structural complexity.

- Structural complexity using Shannon's entropy [Aboutaleb and Monsuez, 2016]

  Measures the structural complexity of a higraph based design using Shannon's entropy.

- Work-effort [Albrecht and Gaffney, 1983]

  Measures work-effort for a software project using function-points and source line of code (SLOC).

- Complexity metrics for service-oriented systems [Zhang and Li, 2009]

  Propose a set of complexity metrics for service-oriented infrastructures.

- Flexibility, complexity, and controllability [Broniatowski and Moses, 2014]

  Measures a systems complexity using existing flexibility metrics. A controllability metrics is proposed since flexibility comes with a cost of decreased control of the system.

- Abstraction level of design patterns [Kubo et al., 2007]

  Measures and indicates the relative abstraction level of each design pattern used.

- Instability and abstractness metrics [Almugrin et al., 2014]

  Measures the instability of classes within a package to conclude which should be abstracted. The abstractness metrics measures the abstractness level of a package.

- Reliability, maintainability, and availability analysis [Gillespie et al., 2012]

  Avionics architecture analysis methodology for predicting the need for redundancy based on failure rate and domain expert opinions.

- Dependency analysis [Nord et al., 2014]

  Measures architectural dependency to reduce cost of safety testing and upgrade in safety-critical systems.

- Predicting maintainability with coupling [Perepletchikov et al., 2007]

  Measures structural coupling of design artefacts to predict maintainability for service-oriented designs.

- Complexity and coupling [Durisic et al., 2013]

  Measures the impact of changes to complexity and coupling in software systems.

- Modifiability [Yau and Chang, 1988]

  Measures the modifiability of a program module.

- Structural distance [Nakamura and Basili, 2005]

  Measures two points in development of software architecture highlight the changes between.

- Modularity designed architecture [Ghasemi et al., 2015]

  Measures modularity of a designed architecture.

- Modularity service architecture [Voss and Hsuan, 2009]

  Decomposes the architectural model to evaluate to what degree of modularity can be used across the system.

- Execution time and performance [Stewart, 2006]

  Measures execution time and performance in embedded real-time systems.

- Execution time and execution jitter [Bril et al., 2008]

  Measures execution time and execution jitter for hard ,single processor, real-time tasks under fixed-priority pre-emptive scheduling.

- Schedulability [Kim et al., 2015; Grigg and Audsley, 1999; Chen and Du, 2015; Vestal, 2007; Yao et al., 2016; Pathan, 2014]

  – Schedulability bound analysis for IMA partitions [Grigg and Audsley, 1999].

  – Schedulability and timing analysis for IMA systems [Chen and Du, 2015].

  – Pre-emptive scheduling analysis of mixed-critical systems with varying execution time [Vestal, 2007].

  - System-level scheduling analysis of mixed-criticality traffics in avionics network [Yao et al., 2016].
  - Fault-tolerant and real-time scheduling analysis for mixed-criticality systems [Pathan, 2014].
  - Timing analysis of mixed-criticality hard real-time applications implemented on distributed partitioned architectures [Marinescu et al., 2012].
  - Response-time analysis for mixed-criticality systems [Baruah et al., 2011].

- Quality [Elhag and Mohamad, 2014]

  Measures the quality of service-oriented design to reduce cost and implementation work.

- Reliability [Gaudan et al., 2008]

  Measures internal risk level of using object-oriented technologies to cut costs.

- Reliability and risk [Schneidewind, 1996]

  Analysis the reliability and risk for safety critical software.

- Reliability - pragmatic approach [Chandran et al., 2010]

  Measures system reliability with different reliability models which take system architecture into account.

- Scalability [Jogalekar and Woodside, 2000]

  Evaluates the scalability of distributed systems.

- Stability [Yau and Collofello, 1985]

  Measures design stability to indicate potential ”ripple effects” when modifying the program in design level phase.

3.2 Selected metrics

A short-list, Appendix B, was created after screening the long-list together with the team of domain experts. The metrics were presented briefly with only a couple of sentences that described their intention. The ones that stood out and caught the attention of the team were discussed prior to any selection.
A smaller study was conducted on the metrics from the short-list to further investigate how they are implemented, what they measure, and whether they could be useful for system architects in the aerospace industry. The metrics from the short-list underwent an additional screening with the team. Eventually, in order to find one to three potential metrics that could be implemented. Three metrics were selected for implementation that showed possible applicability during different stages of product’s life cycle and manageable implementation:

1. Structural complexity using Shannon's entropy [Aboutaleb and Monsuez, 2016]
2. Instability and abstractness [Almugrin et al., 2014]
3. Complexity and coupling [Durisic et al., 2013]

The first metric measures the structural complexity of the system design using a higraph for system modelling and Shannon's entropy for measuring. This metric is still undergoing research but has great potential to provide structural complexity measures during early phases of a project when selecting between different designs for given requirements. In large projects it can identify the most complex architectural design. The second metric, instability and abstractness, is software metric based on R.C. Martin's previous work [Almugrin et al., 2014]. The instability metric measures both instability and stability simultaneously, that is ”stability is the lack or absence of instability” [Almugrin et al., 2014]. Instability of an element e indicates how affected it is when changes are made to other elements in a system. High instability value suggest that element e have many elements that it depends on. The third metric measures complexity and coupling. The metrics ,which are result of a study in the automotive industry, are based on Henry and Kafura's ”fan-in fan-out” concept and Gupta and Chhabra's package coupling metrics (PCM).

Metric 1: Structural complexity using Shannon's entropy

While still under research, the structured complexity metric was selected due to ability to estimate structural complexity of a design which can be of huge importance when comparing different options for the system. The measurement depend on a good model that represents the system of interest. This will give the designers and stakeholders a common picture of system and possibility to point out issues regarding it. The system is modelled using higraph [Aboutaleb and Monsuez, 2016; Harel, 1988; Grossman and Harel, 1997]. The structural complexity is then measured from the higraph model using Shannon's theory of information, entropy.

The higraph is ”a combination and extension of graph and Euler/Venn diagrams[...]. They are formed by modifying Euler/Venn somewhat, extending them to represent Cartesian products, and connecting the resulting 'blobs' by edges or hyperedges.” [Grossman and Harel, 1997]. It other words a higraph is an extended graph that includes depth (hierarchy) and orthogonality (partitioning).

Definition Higraph
A higraph is defined as a tuple $H = (B, E, \rho, \Pi)$ where:

– B is the set of blobs (nodes).

– E is the set of edges.

– ρ is the hierarchy function where each $b \in B$ assigns its set of sub-blobs, $\rho(b)$.

- Π is the orthogonality (partitioning function) defined as $\Pi : B \to 2^{BxB}$. A blob $b \in B$ can partition its sub-blobs into $\pi_1(b), \pi_2(b), ..., \pi_n(b)$ subsets. The union of all partitions $\cup_{i=1}^{n} \pi_i(b)$ should give the same result as $\rho(b)$.

Aboutaleb and Monsuez define five higraph-based transformations that should be used when constructing a higraph; generalization, aggregation, decomposition, refinement, and filtering [Aboutaleb and Monsuez, 2015a]. The transformation produces a model higraph and a type higraph. The model higraph is a representation of the real system that is being designed. The type higraph is created from the model higraph and represents the partitioning within the system. Both higraphs are vital for the metric as they both have attributes that affect the structural complexity.

Definition 1 Generalization

- Let M_Π be a type higraph.
- Let M be a model higraph.
- Let $g : M \to M_\Pi$ a morphism that associates to each element (object, flow, attribute) x of the model higraph M to its type, with M_Π, the model type higraph.
  - $\forall x \in M, g(x) \in M_\Pi$
  - $\forall x \in M, g(\rho(x)) \subset \rho(g(x))$
  - $\forall t \in M_\Pi, g(\Pi_t(x) \subset \rho(t)$

  $M_\Pi = (B; E; \rho; \Pi)$ is a higraph where:
  - $E = 0$
  - $\forall xB, \Pi(x) = \rho(x)$

Definition 2 Aggregation

- Let M be a model higraph.
- Let x be a model node.
- Let y_i be model nodes such that $y \in \rho(x)$.
- The aggregation function f_{agg} maps a set of elements y_i to a single element x:

$$f_{agg} : M \to M \text{ such that } f_{agg}(y_1, ..., y_{|\rho(x)|}) = x$$

  This function is used to represent an object as a black box, without its children. The inverse function is the decomposition function.

Definition 3 Decomposition

– Let M be a model higraph.

– Let x be a model node.

– Let y_i be a model nodes such that $y \in \rho(x)$.

– The decomposition function f_{dec} maps a single element x to a set of elements y_i:

$$f_{dec} : M \rightarrow M \text{ such that } f_{dec}(x) = \{y_1, ..., y_{|\rho(x)|}\}.$$

This function is used to represent an object as glass box, with its children. The inverse function is the aggregation function.

Definition 4 Refinement

– Let M be a model higraph.

– Let x, y be two model nodes. An element x is said to be refined by an element y if x contains y, i.e.:

  - $y \in \rho(x)$

This transformation allows refining an element in the model.

Definition 5 Filtering

– The model views are obtained by filtering the type higraph. Thus, there exists a filtering function that can be applied: *filtering*. The model views are obtained by filtering the type higraph. A filter function is a function $f : M \rightarrow V$, where M_Π is the type higraph and V is a model view, which either preserves nodes in the type higraph or removes them.

  - $V \subset M_\Pi$
  - $\rho(V) \subset \rho(M_\Pi)$

This transformation allows extracting a view containg elements of the model, that are of the same type, and their decomposition.

After creating the higraph it is possible to extract basic metrics such as the size, depth, and width. This will later be useful when measuring the structural complexity.

Definition Size

– The size of a higraph $H = (B, E, \rho, \Pi)$ is defined as $(|B|; |E|)$

- $|B|$ is the number of blobs (nodes).
- $|E|$ is the number of edges.

Definition Depth

– The depth of a higraph $H = (B, E, \rho, \Pi)$ is defined as:

– Let x be a top blob that has no parent and let y be a nested blob such that:

- $\exists k$, such that $y \in p^k(x)$, where k is the number of levels between x and y.
- The depth dp of a blob y is k and the depth Dp of the higraph H is $max_{y \in B}(dp_y)$.

Definition Width

– The width of a higraph $H = (B, E, \rho, \Pi)$ is defined as:

- Let $x \in B$ be a blob.
- The width w of a blob x is $\rho(x)$ and the width W of the higraph H is $max_{x \in B}(w_x)$.

In order to understand how Shannon's entropy is applied to measurement of the structural complexity is it necessary to follow the logic that the authors used. The first step is to define the entropy.

Definition Shannon's entropy
Let X be a set of discrete random variables with values $x_1, x_2, ..., x_n$ with x_i having probability p_i; $(1 < i < n)$. Shannon's entropy H is defined as:

$$H(X) = -\sum_{i=1}^{n} p_i log_2 p_i \tag{3.1}$$

The entropy of a higraph model M is dependent on the number of blobs and edges, the hierarchy and the orthogonality [Aboutaleb and Monsuez, 2016]. However, Shannon's entropy is used to provided information about the complexity. The entropy of the higraph model M is retrieved the following way:

$$H = H_B + H_E + H_\rho + H_\Pi \tag{3.2}$$

Every element in Equation 3.2 measures the complexity separately with Shannon's entropy. The result is then added to represent the total structural complexity.

– H_B (blobs)

$$H_B = H(B) = -log_2(1/|B|) = log_2(|B|) \tag{3.3}$$

– H_E (edges)

$$H_E = H(E) = -2log_2(1/|E|) = 2log_2(|E|) \tag{3.4}$$

Equation 3.4 takes into account the head and the tail of the edge.

– H_ρ (hierarchy)

$$N = \sum_{x \in M} |\rho(x)| \tag{3.5}$$

$$H_\rho = -2log_2(1/|N|) = 2log_2(|N|) = 2log_2(\sum_{x \in M} |\rho(x|) \tag{3.6}$$

– H_Π (partitioning)

Partitioning is performed by grouping blobs of same type into common subset. Measuring the complexity of H_Π is rather complicated and involves creating a new type higraph M_Π that is associated with the original higraph M. The complexity is measured on the new higraph H_Π, $H_\Pi = H(M_\Pi)$.

- Let M_Π be a type higraph.
- Let M be a model higraph.
- Let $g : M \rightarrow M_\Pi$ be a morphism that maps each element from higraph model M to its type in higraph M_Π.
- The new higraph M_Π can be described as $H_\Pi = (B_{M_\Pi}, E_{M_\Pi}, \rho, \Pi)$ with following properties:
  - ◇ $E_{M_\Pi} = 0$, there are no edges.
  - ◇ $\forall x \in B_{M_\Pi}, \Pi(x) = \rho(x)$, all elements are the same type.

The newly created higraph needs to be evaluated the same way as the original higraph model M in Equation 3.2, thus:

$$H_\Pi = H(B_{M_\Pi}) + H(E_{M_\Pi}) + H_\rho(M_\Pi) + H_\Pi(M_\Pi) \tag{3.7}$$

Since the type higraph M_Π is purposely without edges and partitions, Equation 3.7 can therefore be rewritten as:

$$H_\Pi = H(B_{M_\Pi}) + 0 + 2log_2(\sum_{x \in M_\Pi} |\rho(x)|) + 0 \tag{3.8}$$

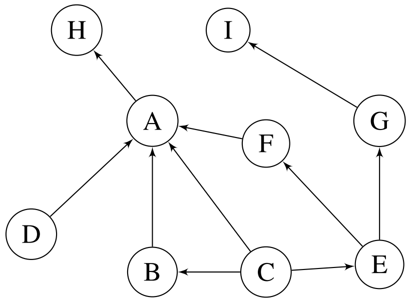


Figure 3.1 Dependency graph G.

Finally, the structural complexity for a system design using higraph is retrieved from the following Equation:

$$H = log_2(|B|) + 2log_2(|E|) + 2log_2(\sum_{x \in B} |\rho(x)|) + log_2(|B_\Pi|) + 2log_2(\sum_{x \in M_\Pi} \rho(x)|) \tag{3.9}$$

Metric 2: Instability and abstractness

Knowing which applications have high instability value can be an indication of sensitivity when modifying the system. The article presents an modified version of R.C. Martin's instability metrics. The authors also present an abstractness metric which following the stable abstractness principle (SAP) should make highly responsible classes abstract since many concrete classes are depended upon them. However, in aircraft system architecture applications have interfaces which they communicate through, making them abstract already by design and not applicable for the task at hand.

The metric is based on software packages but for the sake of our implementation these packages will instead be considered as applications. This can, however, be applied to other domains as well. The chain of measurements needed to get instability value is as follow, "received dependency value" $\rightarrow$ "responsibility value" $\rightarrow$ "relative responsibility value" $\rightarrow$ "instability value".

Prior to any measurement is the creation of a directed dependency graph G = (N,E), where: N = 1,...,n is the set of nodes. E = $\subseteq N_x \times N_y$ is the set of unidirectional edges. An edge, or a dependency, is defined so that the tail node depends on the node which it points at, $N_x \rightarrow N_y$. An simple dependency graph G is constructed and shown in Figure 3.1.

An M $\times$ M adjacency matrix is created where the dependency graph is represented. Every dependency or every edge has the value of 1 and 0 where there is no dependency. Table 3.1 illustrates the adjacency matrix with all of the dependencies. The outgoing dependencies need to be evenly distributed hence normalising the ad-

Table 3.1 An example of an initial adjacency matrix.

	A	B	C	D	E	F	G	H	I
A	0	0	0	0	0	0	0	1	0
B	1	0	0	0	0	0	0	0	0
C	1	1	0	0	1	0	0	0	0
D	1	0	0	0	0	0	0	0	0
E	0	0	0	0	0	1	1	0	0
F	1	0	0	0	0	0	0	0	0
G	0	0	0	0	0	0	0	0	1
H	0	0	0	0	0	0	0	0	0
I	0	0	0	0	0	0	0	0	0

Table 3.2 An example of an normalised adjacency matrix.

	A	B	C	D	E	F	G	H	I
A	0	0	0	0	0	0	0	1	0
B	1	0	0	0	0	0	0	0	0
C	1/3	1/3	0	0	1/3	0	0	0	0
D	1	0	0	0	0	0	0	0	0
E	0	0	0	0	0	1/2	1/2	0	0
F	1	0	0	0	0	0	0	0	0
G	0	0	0	0	0	0	0	0	1
H	0	0	0	0	0	0	0	0	0
I	0	0	0	0	0	0	0	0	0

jacency matrix, Matrix 3.1, before conducting any measurements. Matrix 3.2 is the result of the normalisation. The dependencies are normalised between the values 0 and 1.
Measuring received dependency value of a node:

$$D(N_i) = \sum_{N_j \in N_i} \left(\frac{D_0(N_j) * N_j.Weight(i)}{deg^+(N_j)} \right) \tag{3.10}$$

Here:
$D(N_i)$ = Received dependency value of Node N_i.
D_i = Set of modules depending upon Node N_i.
$D_0(P_j)$ = Initial dependency value which is 1.
$P_j.Weight(i)$ = While the initial dependency value is 1, the algorithm can be adjusted to add weight to a dependant node.
$det^+(N_j)$ = Total number of outgoing dependencies of Node N_j.

Assuming that no weight is added, each node's received dependency value is the sum of the respective column.

The proportional responsibility is based on the following equation:

$$R(N_i) = \frac{D(N_i)}{|D_x|} \tag{3.11}$$

Here:
$R(N_i)$ = Responsibility value of Node N_i.
$D(N_i)$ = Received dependency value of Node N_i.
$|D_x|$ = Total number of the dependent nodes in the system which is all the nodes excluding the sink modules. Sink is a node that does not have any outgoing dependencies.

To normalise the responsibility between 0 and 1, we calculate the relative responsibility as follows:

$$R'(N_i) = \frac{R(N_i)}{R_{max}} \tag{3.12}$$

Here:
$R'(N_i)$ = The relative responsibility of Node N_i.
$R(N_i)$ = Responsibility value of Node N_i.
R_{max} = The maximum responsibility value of node in the system.
Lastly, to measure instability, the following formula is used:

$$I(N_i) = \frac{\sum_{N_j \in D'_i}(1-(R'(N_j) * \partial))}{deg^-(N_i) + deg^+(N_i)} \tag{3.13}$$

Here:
$I(N_i)$ = Instability of Node N_i.
D'_i = Set of nodes that Node N_i depends on.
$R'(N_j)$ = The relative responsibility value of Node N_j.
∂ = The dependency factor with a default value of 0.5. Setting the dependency factor to value of one means that the most responsible node will have no effect on the stability of the dependant node.
$deg^-(N_i)$ = Total number of incoming dependencies of Node N_i.
$deg^-(N_i)$ = Total number of outgoing dependencies of Node N_i.

Metric 3: Complexity and coupling

Metric 3 is not a newly defined metric but rather improved version of existing design metrics. The authors behind Metric 3 conducted a metric study in the automotive industry and concluded that two metrics were significantly reliable for minimising cost and complexity in development of system architecture. Henry and Kafura's complexity metric ”fan-in fan-out” to measure complexity and Gupta and Chabbra PCM for measuring coupling [Henry and Kafura, 1981; Yourdon and Constantine,

Table 3.3 Example of the weights used in the automotive industry. Source [Durisic et al., 2013].

type of signal	weight
intra-sub-system	1
inter-sub-system	1.3
inter-domain	1.8

1979; Gupta and Chhabra, 2009]. The authors also define two software development views of the system; logical view and deployment view. The logical view is used for software design whereas deployment view is used for mapping software components to hardware. All the metrics can be used in both views except for the weighted equation which differs.

Complexity. Henry and Kafura defined their complexity metric the following way:

$$C(i) = (fin(i) \times fout(i))^2 \tag{3.14}$$

"fin" represents total number of modules calling module *i* and "fout" represents total number of calls module *i* is making to other modules. It is then squared to punish any kind of bidirectional dependency.

Complexity in Metric 3 takes complexity attributes such as hierarchical levels of signals and timing constraints into account when modifying "fan-in fan-out" metric. The exponent is removed, due to unreasonable amplifications, and the modified metric is defined as:

$$C(i) = (cin(i) \times cout(i)) \tag{3.15}$$

$$C(n) = \sum_{i=1}^{n} C(i) \tag{3.16}$$

Distributed software systems in automotive industry often have different suppliers and having components with two-way dependencies is therefore far more dangerous than one way and thus, the complexity is zero if there is only one way dependency.

A graph, representing the system dependency, is created and edges are weighted in correlation to what type of signal is transmitted between the nodes in the real system e.g. intra-sub-system signals, inter-sub-system signals, and inter-domain signals. The weight equation, Equation 3.17, is define as:

$$w(s,s') = \sum_{i=1}^{l} type(sig_{s \to s'}(i)) \tag{3.17}$$

The equation defines the weight of an edge as the sum of all signals *l* sent between component *s* and *s'* with corresponding weight to signal type *i*. An example of the value used in the study in the automotive industry is presented in Table 3.3.

The weighted equation for deployment view focuses on intra-computer and inter-computer signals with timing constraints. *MaxAge* is the maximum allowed time a signal can get from s to s'.

$$w(s,s') = \sum_{i=1}^{l} type(sig_{s\to s'}(i)) \times \left(1.5 - \frac{MaxAge(sig_{s\to s'}(i))}{2000ms}\right) \tag{3.18}$$

The graph is complete when all of the edges have weights related to their type of signal. The modified complexity metric, Equation 3.15, is then redefined to include signal weight. Equations 3.20 is inserted into Equation 3.15 to form the Equation 3.21.

$$\begin{aligned} cout(s) &= \sum_{s',s''\in S\wedge(s',s'',w)\in D\wedge s'=s} w(s',s'') \\ cin(s) &= \sum_{s',s''\in S\wedge(s',s'',w)\in D\wedge s''=s} w(s',s'') \end{aligned} \tag{3.20}$$

The *cout* and *cin* of a component s is therefore sum of all incoming weighted dependencies multiplied with the sum of all outgoing weighted dependencies, Equation 3.21.

$$C(s) = \sum_{s',s''\in S\wedge(s',s'',w)\in D\wedge s'=s} w(s',s'') \times \sum_{s',s''\in S\wedge(s',s'',w)\in D\wedge s''=s} w(s',s'') \tag{3.21}$$

Every component's complexity is then summed to get a total for the system.

$$C(n) = \sum_{s\in S} C(s) \tag{3.22}$$

Coupling. PCM calculates coupling between two packages of the same hierarchical level i [Gupta and Chhabra, 2009]. It is the dependencies between components within the different packages that is measured. Equation 3.23 defines coupling between two packages p_a^l and p_b^l as total number of depend upon relationship r between components e_i and e_j of all hierarchical levels added with respective for depended relationships.

$$Coup(p_a^l, p_b^l) = \sum_{i=1}^{n} \sum_{j=1, j\neq i}^{m} r(e_i^{l+1}, e_j^{l+1}) + \sum_{j=1}^{n} \sum_{i=1, i\neq j}^{m} r(e_j^{l+1}, e_i^{l+1}) \tag{3.23}$$

PCM value for package P_a^l is defined in Equation 3.24. The value is retrieved by through coupling, Equation 3.23, between package P_a^l and other packages of same hierarchical level.

$$PCM(P_a^l) = \sum_{b=1\wedge b\neq a}^{t} Coup(P_a^l, P_b^l) \tag{3.24}$$

Inspired by PCM, the authors have modified the Equation 3.23 to instead measure the weighted dependencies between the packages. In order to realise this, the earlier created graph with the weighted edges is used. Figure 3.2 illustrates two packages, *level i*, that contain applications and sub-packages, denoted *level i+1*. The sub-package has an application which belongs to *level i+1*.

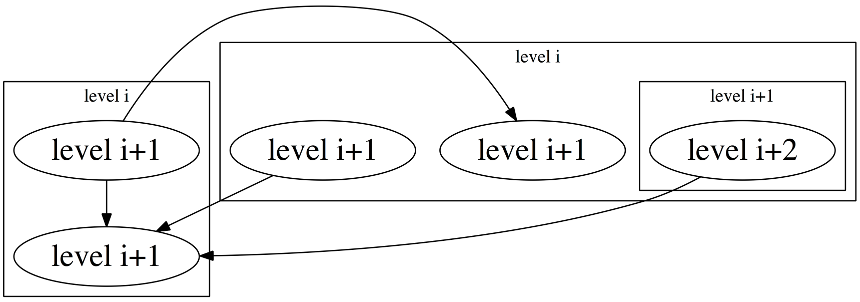


Figure 3.2 Two packages with applications and sub-packages on different hierarchical levels.

$$Coup(p,p') = \sum_{s,s' \in S \wedge (s,s',w) \in D \wedge s \in p \wedge s' \in p'} w(s,s') + \sum_{s,s' \in S \wedge (s',s,w) \in D \wedge s \in p \wedge s' \in p'} w(s,s') \tag{3.25}$$

By using the constructed graph with weighted edges, the PCM equation, Equation 3.23, is redefined to Equation 3.25 that takes the weighted edges w as relationship between components instead of r. The PCM value for a package is obtained through Equation 3.26 which sums all coupling values between package p and other packages of same hierarchical level.

$$PCM(p) = \sum_{p',p'' \in P \wedge p = p'} Coup(p',p'') \tag{3.26}$$

4

Implementation and Validation

4.1 Prerequisites

IMA architectures are highly complex with dependencies throughout the whole system. To model the system with high detail is complicated and requires knowledge of the product being developed. The system models in the thesis are inspired by IMA architecture and should be treated as example rather than guidelines when implementing the metrics. The metrics are implemented in high level programming language. It should be emphasised that language has limited programming environment and that the implementations can be optimised using a more appropriate programming language. To illustrate the implementation and usage, a small scaled system, Figure 4.1, is designed to assist and provide better understanding what the metrics are measuring throughout the chapter. The theory should also be applicable to large scale systems.

Prior to any implementation, the selected metrics were assessed to establish which input values are necessary to produce measurements. All of the metrics rely on graph based solution on which measurements are conducted. Metric 2 and Metric 3 use directed graphs for representing the dependencies between system components. Metric 1 uses higraph when modelling the system. In order to avoid possible misrepresentations of the system between different graphs, the solution is to rely on one graph which will satisfy all three metrics. Having the higraph and directed graph complement each other enables all three metrics to share common input values. The edges defined in higraph could be weighted in order to satisfy Metric 3 properly. The edges in Metric 3 are weighted in regard to type of signal. The weights should be set by system architects in cooperation with stakeholders and engineers, however, for this implementation no weights are defined. Every edge has a weight of 1 and the directed graph can be used in all metrics without any modification. Modelling the system is done with higraph and the dependencies are defined with a directed graph. The graphs can be represented through two dimensional matrices, meaning that all

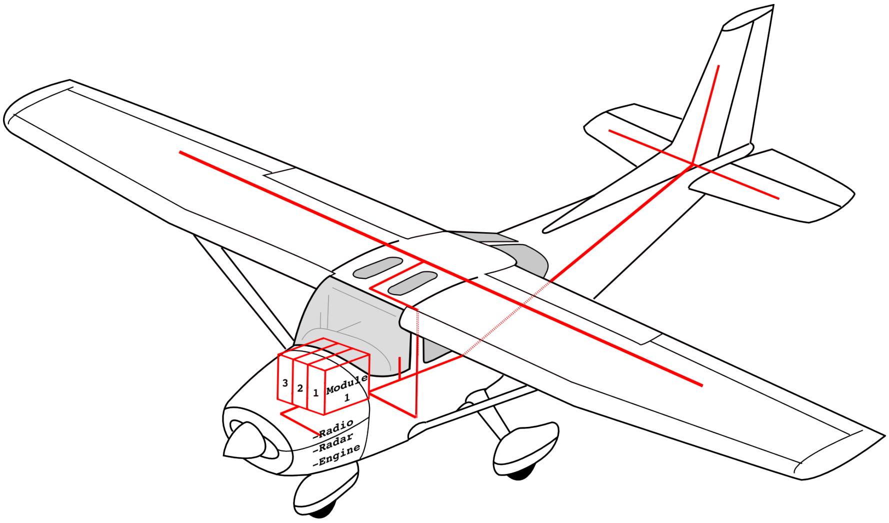


Figure 4.1 Graphical view how the test bed can be installed in an aircraft.

metrics have a valid input parameter. Metric 3 needs additional information in order to measure coupling; module list. Due to limited programming environment, components within a package are represented through a list.

4.2 Test bed

For the purpose of visualising the system and its functions, Figure 4.1 illustrates the test bed and how a system architecture can be designed for a small aircraft. The three modules, computers, run a total of 10 applications. As an example, Module 1 could be responsible for the controls, Module 2 for radio, radar, and displays, and Module 3 for the engine. The modules are running applications on six separate partitions. The system architecture of the aircraft is first modelled with a tree graph, Figure 4.2. The graph shows which applications are assigned to respective partitions.

By following the higraph-based transformation, the tree can be transformed into a higraph, Figure 4.3. As the partitions in higraph are not blobs they are not modelled in the tree graph, Figure 4.2. The type higraph can not be transformed the same way as the model higraph, from a tree graph. Type higraph is a product of the model higraph which defines encapsulation of blobs. The corresponding tree graph for a type higraph would likely have the layout presented in Figure 4.4.

Both higraph and type higraph must be transformed into matrix form, Table 4.1 respective Table 4.2, to conduct measurement.

Metric 2 and Metric 3 are relevant only if there are dependencies between applications. 16 unique edges were randomly generated, spanning through the whole

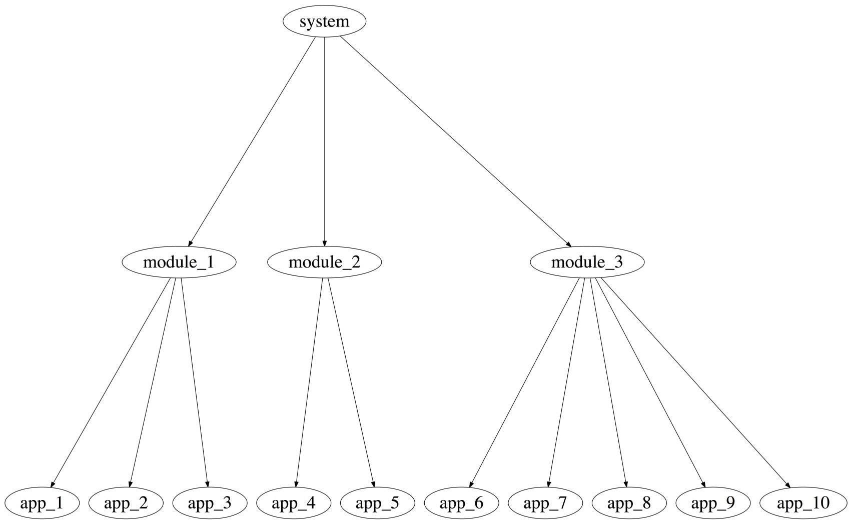


Figure 4.2 Test bed. Graph representation of the system.

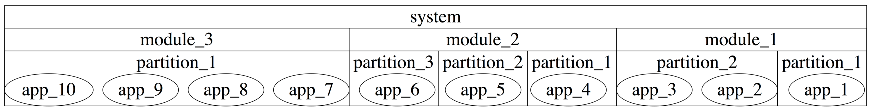


Figure 4.3 Test bed. Higraph representation of the system model.

system. In comparison to IMA architecture development, the domain experts suggest a producer-consumer design pattern since the applications are executed in a hard real-time environment with strict constraints. Part of the applications in the test bed are considered consumers; they are depended upon other applications. Another part are producers that have applications that depend upon them. Exceptional applications are both producers and consumers; they depend on other applications while they are depend upon. The system model with dependencies, Figure 4.5, correspond to the system dependency matrix in Table 4.3. From the defined system the following data is extracted:

- System model with dependencies, Figure 4.5, provides the dependency matrix, Table 4.3.
- The model higraph, Figure 4.3, provides both a matrix representation of the higraph and the type higraph model, Table 4.3 respective Table 4.2.

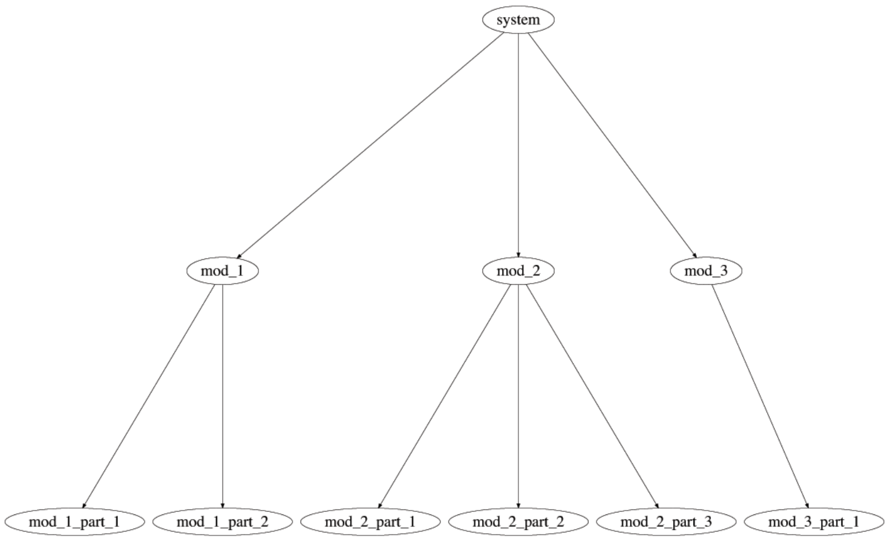


Figure 4.4 Test bed. Tree representation of the type higraph.

Table 4.1 Test bed. Matrix representation of the model higraph of the system system.

Parent \ Child	system	module 1	module 2	module 3	app 1	app 2	app 3	app 4	app 5	app 6	app 7	app 8	app 9	app 10
system	0	1	1	1	0	0	0	0	0	0	0	0	0	0
module 1	0	0	0	0	1	1	1	0	0	0	0	0	0	0
module 2	0	0	0	0	0	0	0	1	1	1	0	0	0	0
module 3	0	0	0	0	0	0	0	0	0	0	1	1	1	1
app 1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
app 2	0	0	0	0	0	0	0	0	0	0	0	0	0	0
app 3	0	0	0	0	0	0	0	0	0	0	0	0	0	0
app 4	0	0	0	0	0	0	0	0	0	0	0	0	0	0
app 5	0	0	0	0	0	0	0	0	0	0	0	0	0	0
app 6	0	0	0	0	0	0	0	0	0	0	0	0	0	0
app 7	0	0	0	0	0	0	0	0	0	0	0	0	0	0
app 8	0	0	0	0	0	0	0	0	0	0	0	0	0	0
app 9	0	0	0	0	0	0	0	0	0	0	0	0	0	0
app 10	0	0	0	0	0	0	0	0	0	0	0	0	0	0

- The model higraph, Figure 4.3, provides two additional lists; the partition list, Table 4.4, and the module list, Table 4.5. The focus in this thesis is on module level which means that the partition list, Table 4.4, is omitted.

Table 4.2 Test bed. Matrix representation of the type higraph of the system.

Parent \ Child	system	module 1	module 2	module 3	mod 1-part 1	mod 1-part 2	mod 2-part 1	mod 2-part 2	mod 2-part 3	mod 3-part 1
system	0	1	1	1	0	0	0	0	0	0
module 1	0	0	0	0	1	1	0	0	0	0
module 2	0	0	0	0	0	0	1	1	1	0
module 3	0	0	0	0	0	0	0	0	0	1
mod 1-part 1	0	0	0	0	0	0	0	0	0	0
mod 1-part 2	0	0	0	0	0	0	0	0	0	0
mod 2-part 1	0	0	0	0	0	0	0	0	0	0
mod 2-part 2	0	0	0	0	0	0	0	0	0	0
mod 2-part 3	0	0	0	0	0	0	0	0	0	0
mod 3-part 1	0	0	0	0	0	0	0	0	0	0

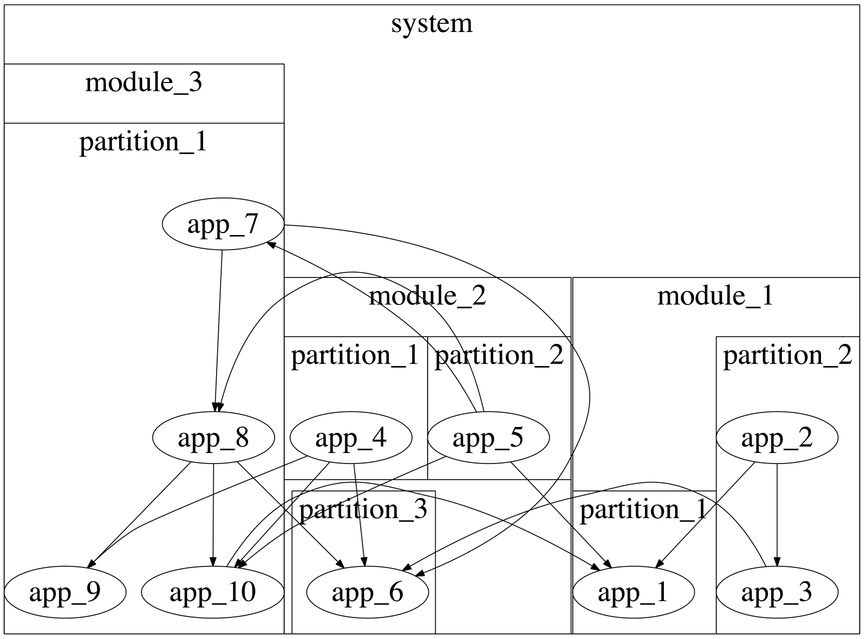


Figure 4.5 Test bed. Higraph representation of the system model with edges.

4.3 Approach

Metric 1

Measuring the structural complexity with Shannon's entropy requires the system model matrix, Table 4.1, system type matrix, Table 4.2 and dependency matrix, Table 4.3. Equation 3.2 defines that entropy is calculated for every element of the higraph definition and Equation 3.8 defines how to calculate the entropy of the type higraph, H_{Π}. Algorithm 1 defines how Metric 1 is implemented. The system model matrix provides the total amount of blobs and dependency matrix provides the total amount of edges. Blobs and edges are calculated by summing the system model

Table 4.3 Test bed. Matrix representing the dependencies between applications.

Application \ Depend on	app 1	app 2	app 3	app 4	app 5	app 6	app 7	app 8	app 9	app 10
app 1	0	0	0	0	0	0	0	0	0	0
app 2	1	0	1	0	0	0	0	0	0	0
app 3	0	0	0	0	0	1	0	0	0	0
app 4	0	0	0	0	0	1	0	0	1	1
app 5	1	0	0	0	0	0	1	1	0	1
app 6	0	0	0	0	0	0	0	0	0	0
app 7	0	0	0	0	0	1	0	1	0	0
app 8	0	0	0	0	0	1	0	0	1	1
app 9	0	0	0	0	0	0	0	0	0	0
app 10	1	0	0	0	0	0	0	0	0	0

Table 4.4 Testbed. Partition list.

partition	**applications**
mod 1-part 1	app 1
mod 1-part 2	app 2, app 3
mod 2-part 1	app 4
mod 2-part 2	app 5
mod 2-part 3	app 6
mod 3-part 1	app 7, app 8, app 9, app 10

Table 4.5 Test bed. Module list.

module	**applications**
module 1	app 1, app 2, app 3
module 2	app 4, app 5, app 6
module 3	app 7, app 8, app 9, app 10

matrix respective dependency matrix. Calculating total number hierarchies requires a call to the custom function *hierarchy*. The *hierarchy* function can be viewed as summing the amount of sub-blobs each blob has. This is implemented by checking for each row which column number has the value of, 1. When discovered, the entire row of the column number is summed and added to hierarchy result. Measuring the entropy on the elements in the type higraph is based on same technique as higraph but without taking edges and H_Π into account.

Using the test bed, the structural complexity is generated by calculating the entropy for the higraph system model and the entropy for the type higraph. The result from Metric 1 using test bed values is the following:

- $|B|$ - 13 blobs.
- $|E|$ - 16 edges.
- $\sum|\rho(x)|$ - 10 hierarchy.

Algorithm 1 Metric 1

```
procedure STRUCTURE COMPLEXITY
    $(system\_matrix, system\_type\_matrix, dependency\_matrix)$
    $blobs \leftarrow \sum(system\_matrix)$
    $edges \leftarrow \sum(dependency\_matrix)$
    $rho \leftarrow hierarchy(system\_matrix)$

    $blobs_\pi \leftarrow \sum(type\_matrix)$
    $rho_\pi \leftarrow hierarchy(type\_matrix)$

    $H \leftarrow log_2(|blobs|) + 2log_2(|edges|) + 2log_2(|rho|) + log_2(|blobs_\pi|) + 2log_2(|rho_\pi|)$

    return $H$

function HIERARCHY(matrix)
    for each row $i$ in $matrix$ do
        for each column $j$ in $matrix$ do
            if $matrix_{(i,j)}$ equals 1 then
                $value \leftarrow value + \sum(matrix_{row(j)})$
    return $value$
```

- (Type higraph) $|B|$ - 9 blobs.
- (Type higraph) $\sum|\rho(x)|$ - 6 hierarchy.

The result avoid a result of negative infinity, $log_2(0)$ equals $-inf$, the specific parameter which holds the value of 0 must be excluded from the equation. The values obtained from the input parameters are inserted into Equation 3.9 and the structural complexity is retrieved:

$$H = log_2(13) + 2log_2(16) + 2log_2(10) + log_2(9) + 2log_2(6) \tag{4.1}$$

$$H = 26.6841 \tag{4.2}$$

Metric 2

While Metric 2 consists of two metrics, instability and abstractness, only the former is relevant in this thesis which is defined in Algorithm 2. The latter metric measures the abstractness of a package and as mentioned earlier, communication between applications in the IMA architecture is performed through abstract interfaces that each application own. Hence, the abstractness metrics is redundant. The only data necessary to calculate the instability of the applications is the dependency matrix e.g. the test bed Table 4.3. Prior to performing any calculations on the dependency

Algorithm 2 Metric 2

```
procedure INSTABILITY($dependency\_matrix$)
    $normalised\_matrix \leftarrow normalise\_dependency(dependency\_matrix)$
    $received\_dependency \leftarrow \sum_{columns}(normalised\_matrix)$

    $D_x \leftarrow \lceil \sum(received\_dependency) \rceil$
    for each value $val$ in $received\_dependency$ do
        $proportional\_responsibility \leftarrow val/D_x$

    $R_{max} \leftarrow max(proportional\_dependency)$
    for each value $val$ in $proportional\_responsibility$ do
        $relative\_responsibility \leftarrow val/R_{max}$

    for each application $app$ in $dependency\_matrix$ do
        for each application $app\_dep$ that depends on application $app$ do
            $numerator \leftarrow$
            $\sum(1 - (relative\_responsibility_{app\_dep} \times dependency\_factor))$
        $instability_{app} \leftarrow numerator / \sum dependencies_{app}$
    return $instability$

function NORMALISE_DEPENDENCY(matrix)
    for each row $i$ in $matrix$ do
        $new\_matrix \leftarrow matrix_{row(i)} / \sum(matrix_{row(i)})$
    return $new\_matrix$
```

matrix, it needs to be normalised between value 0 and 1. The custom function *normalise_dependency* takes the dependency matrix as input parameter, calculates the sum of a row and divides each column on the specific row with the row sum. The normalise matrix is depict in Table 4.6.

Received dependency. As mentioned in Chapter 3, the instability value is acquired through multiple equations. The received dependency value, Equation 3.10, is cal-

Table 4.6 Test bed. Matrix representing evenly distributed dependencies between applications.

Application \ Depend on	app 1	app 2	app 3	app 4	app 5	app 6	app 7	app 8	app 9	app 10
app 1	0	0	0	0	0	0	0	0	0	0
app 2	1/2	0	1/2	0	0	0	0	0	0	0
app 3	0	0	0	0	0	1	0	0	0	0
app 4	0	0	0	0	0	1/3	0	0	1/3	1/3
app 5	1/4	0	0	0	0	0	1/4	1/4	0	1/4
app 6	0	0	0	0	0	0	0	0	0	0
app 7	0	0	0	0	0	1/2	0	1/2	0	0
app 8	0	0	0	0	0	1/3	0	0	1/3	1/3
app 9	0	0	0	0	0	0	0	0	0	0
app 10	1	0	0	0	0	0	0	0	0	0

culated by summing each column of the normalise matrix in Table 4.6. The result of received dependency is obtained in Table 4.7.

Proportional responsibility. The proportional responsibility, Equation 3.11, uses the values retrieved from received dependency equation. Every value is divided by the total number of dependent applications, D_x. Dependent applications in the dependency graph, Figure 4.6, are those who are not defined as sink and not dependent on other applications. The result of the proportional responsibility is obtained in Table 4.7.

Relative responsibility. The relative responsibility, Equation 3.12, divides each value with the highest proportional responsibility value, R_{max}. The R_{max} value for the test bed system system is 0.3095. Table 4.7 presents the result for the relative responsibility.

Instability. The instability, Equation 3.13, can now be calculated. For demonstration purposes the value of the dependency factor, ∂, is set to 0.5. ”Setting the dependency factor (∂) to one means that depending on the most responsible package will have no effect on the stability of the dependent package” [Almugrin et al., 2014]. Algorithm 2 divides the instability equation into two steps because it simplifies the measurement. The instability equation, Equation 3.13, has a sum in the numerator which is then divided by all of dependencies, both outgoing and incoming, in consideration for the application being measured. Outgoing and incoming dependencies are calculated from the test bed dependency matrix, Table 4.3. Outgoing dependencies are equivalent to summing the rows and incoming dependencies are equivalent to summing the columns for each application. The result of instability, and previous, equations are presented in Table 4.7.

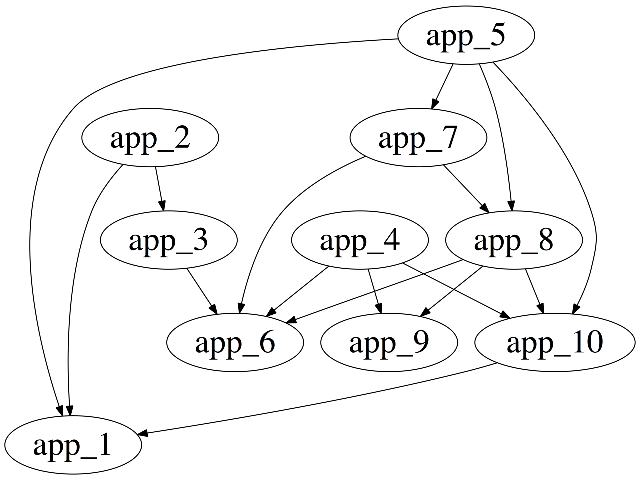


Figure 4.6 Test bed. Dependency graph for Metric 2.

Table 4.7 Test bed. Presenting all result from equation leading up to instability value.

equation \ application	1	2	3	4	5	6	7	8	9	10
received dependency	1.75	0	0.5	0	0	2.1667	0.25	0.75	0.6667	0.9167
proportional responsibility	0.25	0	0.0714	0	0	0.3095	0.0357	0.1071	0.0952	0.1310
relative responsibility	0.8077	0	0.2308	0	0	1.00	0.1154	0.3462	0.3077	0.4231
instability	0	0.7404	0.25	0.7115	0.7885	0	0.4423	0.4269	0	0.1490

Metric 3

The third metric consists of two metrics; complexity and coupling. The two metrics need the test bed dependency matrix. Additional data is necessary for coupling since if can be applied to different hierarchy levels of the system. The partition and module list for the test bed, Table 4.4 and Table 4.5, provide the information which enables the measurement. Algorithm 3 and Algorithm 4 define the implementation for complexity and coupling which only set out to measure complexity between applications and PCM between modules.

Complexity. The modified Henry and Kafura's "fan-in fan-out" complexity metric, Equation 3.15, multiplies all outgoing with all incoming dependencies of each

Table 4.8 Test bed. Complexity.

metric \ application	1	2	3	4	5	6	7	8	9	10
complexity	0	0	1	0	0	0	2	6	0	3

application. The automotive industry considers bidirectional dependencies to cause more complexity than one directional, thus, the value is zero if either incoming or outgoing dependencies are zero [Durisic et al., 2013]. Similar to Metric 2, the outgoing dependencies from each application are retrieved by summing the rows and incoming by the columns of the dependency matrix, Table 4.3.

Algorithm 3 Metric 3

```
procedure COMPLEXITY($dependency\_matrix$)
    for each application $app$ in $dependency\_matrix$ do
        $complexity_{app} \leftarrow \sum dependencies_{outgoing} \times \sum dependencies_{incoming}$
    return $complexity$
```

By summing row and column of the square dependency matrix and storing the values in two different vectors, the complexity can be measured in one for-loop by multiplying the same indices from each vector. Complexity measurement for the system is presented in Table 4.8.

Coupling. The metric is based on Gupta Chhabra's PCM and while Metric 3 defines a weighted value between the dependencies, for the implementation in the thesis, the weight is set to 1 regardless of type of signal. The weights should be set by system architects and developers to highlight critical connections. The values are acquired through tests and experiments.

In this implementation, the measurements are conducted on module level using Table 4.5. However, by using partition list, Table 4.4, PCM can be measured on partition level. The first column in Table 4.5 is removed so only the application numbers are left on each row.

Algorithm 4 Metric 3

```
procedure COUPLING($dependency\_matrix, module\_list$)
    for each application $app$ in module $mod$ from $module\_list$ do
        for each application $app\_dep$ from $dependency\_list$ not in $mod$ do
            $coupling_{mod} \leftarrow \sum dependencies_{(app, app\_dep)}$
    return $coupling$
```

The coupling value for each module is acquired by calling the translated PCM equation, Equation 3.25. The result is displayed in Table 4.9.

Table 4.9 Test bed. Coupling.

metric \ module	1	2	3
coupling	3	9	8

Table 4.10 Test bed. Summary of the result.

metric	*description*	**value**
1	structural complexity	26.6841
2	$\sum$ application instability	3.5087
3	$\sum$ complexity	12
3	$\sum$ coupling (modules)	20

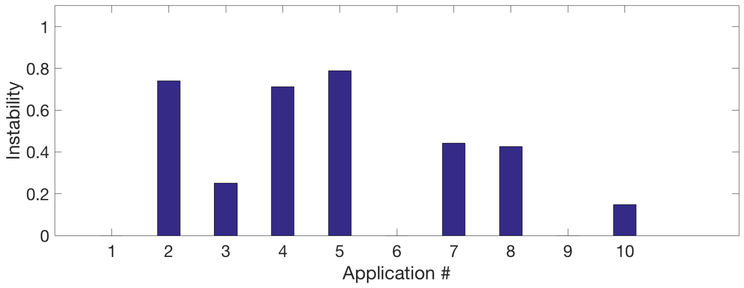


Figure 4.7 Test bed. Visualising instability through bar graph.

4.4 Visualisation

Results from the measurements are acquired in numerical format. Small scale systems, e.g. test bed, are comprehensible. Creating larger systems and more complex requires the metrics to assist the analysis to become more effective and efficient. Visualising the results through charts and graphs can provide the users with better overview of the system and manageably identify areas of concern. Visual representation of the results can be useful but Metric 1, which only outputs one value, is better represented numerically. This section provides examples of visualisation techniques for different metrics. Metric 2 and Metric 3 are visualised through graphs and charts. Metric 1, which is presented numerically in Table 4.10, will not be visualised graphically. The result from all metrics, Equation 4.2, Table 4.7, Table 4.8, and Table 4.9 are summarized in Table 4.10.

Instability value has a range between $0 \leq x \leq 1$. The result from instability measurement is displayed in Figure 4.7. The numbers in x-axis correspond to application with same value, e.g. *app_#*. Complexity value has a range between $0 \leq x \leq \infty$. The result from complexity measurement is displayed in Figure 4.8. Like instabil-

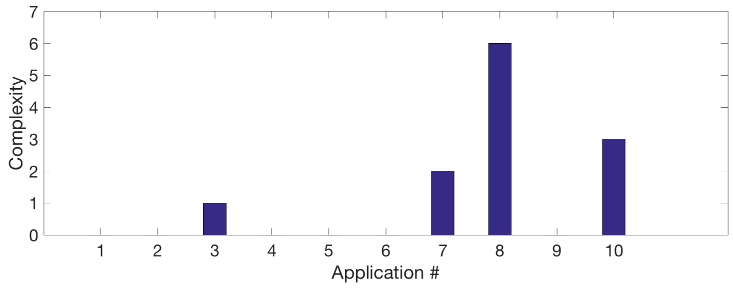


Figure 4.8 Test bed. Visualising complexity through bar graph.

ity, the numbers in x-axis correspond to applications. Instability and complexity are evaluating applications based on the dependency between them and while the results of the metrics can be display in individual figures, combining them into one provides a possibility for effective assessment. Figure 4.11 represents the instability and complexity result.

Coupling can be display the same way as instability and complexity, Figure 4.10. However, the bar chart does not display between which modules coupling is the highest. Figure 4.11 solves the issue by having a graph with weighted edges between package of interest. The weights are the actual coupling values between two entities. While the Figure 4.10 and Figure 4.11 are showing coupling on module level, it can easily be altered to measure coupling between other system components of same hierarchical level.

4.5 Interpretation

Understanding the metrics and interpreting the result is an important corner stone of the validation process. Knowing what is being measured and interpreting the result can help users evaluate whether the metrics are appropriate for the task at hand. Metric 1 is currently being researched. Although couple of examples have been presented in the articles, none of them succeed in presenting explanation how the metric should be interpreted [Aboutaleb and Monsuez, 2016; Aboutaleb and Monsuez, 2015b; Aboutaleb and Monsuez, 2015a]. Metric 2 is a modified C.R. Martin metric. Martin defines stability not as ”likelihood of the package changing” but as a package that other packages depend upon. Making changes in a stable package may affect many other packages. The instability metric inverts the logic and points out which packages are affected when others are modified. The higher the instability value, the higher are the chances that the package will be subject to changes. Values that have zero instability value are considered very stable packages. Metric 3 measures both complexity and coupling. The metrics are effective when comparing between releases. An increase in complexity or coupling is an alert for the users that area is in need of attention. However, good knowledge about the system is required because more complexity or coupling does not necessary equals problem.

4.6 Validation

Theoretical & empirical validation

Majority of the researched metrics in this thesis have been validated both theoretically and empirically by other authors. This holds true for both Metric 2 and Metric 3. However, Metric 1, which measures the structural complexity with Shannon's entropy, is recently defined and still being researched. Understanding and interpreting the result has not been described and although the authors of the metric have defined the rules for modelling and measuring, there has not been conducted any theoretical nor empirical validation [Aboutaleb and Monsuez, 2016; Aboutaleb and Monsuez, 2015b; Aboutaleb and Monsuez, 2015a].

Metric 2 is based on R.C. Martin's metrics for instability and abstractness. Research projects have validated R.C. Martin's metrics and principles, both theoretical and empirical. The modified version has been empirically validated through a case study on an open source system performed by the authors [Almugrin et al., 2014]. Metric 3 use Henry and Kafura *fan-in fan-out* complexity measure and Gupta and Chhabra *PCM*. Both of the metrics satisfy the theoretical validation defined [Briand et al., 1996]. The empirical validation was conducted through case study during a real project in the automotive industry which offered feedback through workshops and interviews with system architects, designers, and integration testers [Durisic et al., 2013].

Cohesion metric was omitted in Metric 3 by the authors of the article due to the fact that modules provided by the suppliers to original equipment manufacturer (OEM) had "platform specific executable code", meaning that the necessary source code needed for the analysis was usually not available to the OEMs [Durisic et al., 2013].

Empirical validation of the selected metrics

This section proposes how empirical validation of the selected metrics for usage in aerospace industry should be conducted. The resemblance in design and development of systems architecture between the automotive and aerospace industry suggests similar approach when validating the selected metrics for usage in the aerospace industry. Metric 3 was empirically validated in the automotive industry according to the guidelines defined in [Fenton and Pfleeger, 1997]. Therefore, the method proposed is highly influenced by the approach used for Metric 3 [Durisic et al., 2013].

1. *Extract the data from one version of the architecture and check its validity manually (to avoid measurement errors).*

2. *Wait for a relevant period of time (decided based on the project schedule/progress) until the number of changes in the architecture accumulates.*

3. *Extract the data from another version of the architecture and check its validity manually.*

4. *Compare the extracted data for both versions and list the changes.*

5. *Interview technical experts in the domains of software architecture and testing in order to check whether the changes listed reflect the changes made in the architecture with respect to Metric 1, Metric 2, and Metric 3.*

The time frame for conducting the empirical validation should span over sufficiently long period of time in order to collect samples during different stages of the development. The metrics rely on a higraph model to execute the measurement. It is therefore necessary for the extracted data, from the architecture, to be representable in a higraph to insure measurement result. The extraction may be performed using custom software tool that collects the information from the primary development environment.

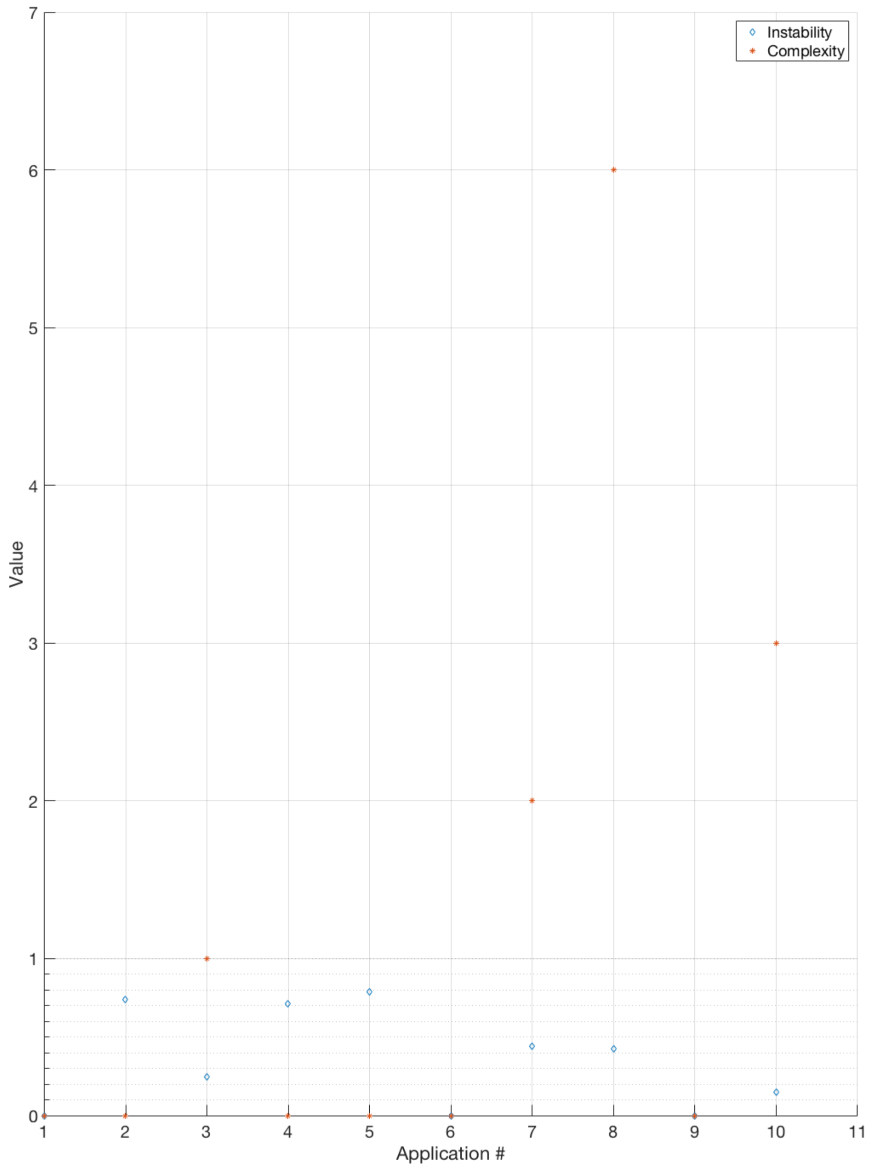


Figure 4.9 Test bed. Example of how to visualise instability and complexity.

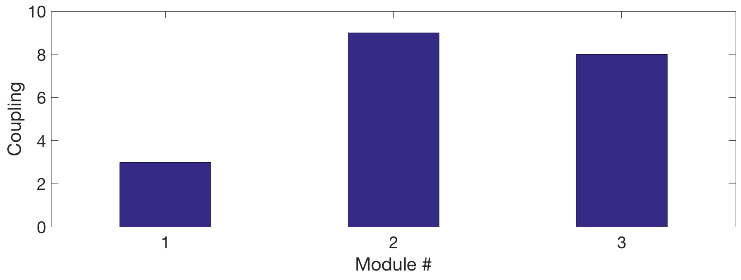


Figure 4.10 Test bed. Example of how to visualise instability and complexity.

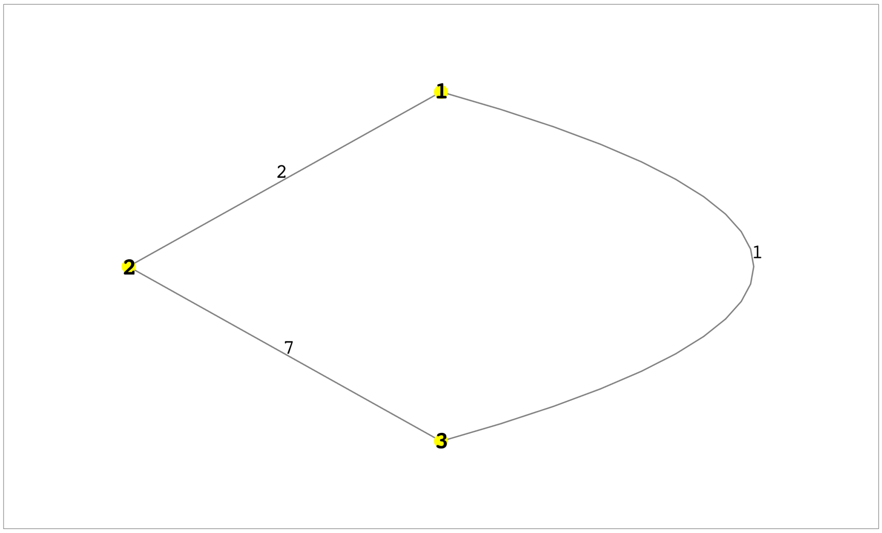


Figure 4.11 Test bed. Coupling visualised.

5 Conclusion and future work

5.1 Conclusion

Systems architecting consumes a small budget of a product's life cycle. Deciding the right architecture can save development cost whereas late modifications are highly expensive. Finding metrics that are used within systems architecting proved to be difficult task. Architects rely much on their experience from past projects, system knowledge, and design patterns. These are all important factors when developing new products and impossible to measure with any metric. This thesis narrows down three potential metrics that can be used by system architects and stakeholders to follow trends throughout a product's life cycle – from conceptual and design to maintenance and support phase – in order to achieve characteristics, e.g. interchangeability, extensibility, and modularity, when developing IMA architecture.

All three metrics rely on a system model that portrays the system viewed from system architects and stakeholders perspective. In the thesis, the system was modelled using higraph from Metric 1 – an extended graph which includes hierarchy and partitioning. The model should be intuitive and only incorporate systems that are of relevance for system architects and stakeholders. The higraph has the benefits of representing dependencies, hierarchy, and partitions with a graph and although no weights were used for Metric 3 it is important to point out that the weighted dependencies are not represented. Additionally, the coupling metric of Metric 3 requires extraction of information from the higraph to differentiate in which package each component belongs.

The new and none mature Metric 1 measures the structural complexity using the higraph which can provide good insight how complex the system is modelled during conceptual and design phases before any work is begun. Different designs can be compared to select the best suited for the requirements. Necessary information on interpreting the result is missing from the articles, likewise theoretical and empirical validation, leaving the users guessing and questioning their designs. A design with higher structural complexity is not equivalent with bad design. The complexity might be justifiable for the specific system and therefore it is important that system architects manually analyse before taking any decisions if they were to use it.

Metric 3 is the only metric found that exists within the field of systems architecting in the industry. It focuses on measuring the effect of changes to its architectural properties and is applied after significant changes in the architecture during development. The metric has both been theoretically and empirically validated. One drawback experienced during the implementation is the complexity metric. In the automotive industry the complexity metric is 0 if a component has unidirectional dependencies, i.e. only incoming or outgoing dependencies. This suits the industry because that is the desired flow of communication between components, meaning that bidirectional connections are more likely to cause problems. However, the system architecture in the aerospace industry, especially with IMA architecture, handles more data which is shared between sub-systems. Component with high degree of depend upon components should also be considered complex since minor changes could also effect large parts of the system, causing ripple effect.

Metric 2 which is a pure software metric is used to highlights components that are likely to be affected by modifications in the system. Metric 2 measures instability but with a smaller modification it can be adjusted to highlight the most stable and responsible components as they are more likely to produce ripples in the system when modified. While the result from the measurement varies between 0 and 1 it is not obvious how it can be applied to support system architects and stakeholders. Possible application could be to use this metric in cooperation with the complexity metric in Metric 3 to highlight the behaviour of the components with complexity value of 0.

Metrics within the systems architecting is relatively unexplored research field unlike e.g. software architecting which has made great progress over the years. It is possible that further software metrics could be used in systems architecting but which ever emerges, system architects should not base their decisions solely on the result of the measurements. The metrics should only provide support whereas knowledge, experience, and design patterns should still be used when analysing the system and making decisions. The selected metrics need to be validated in the aerospace industry over a longer period of time before any decision can be made whether or not they are relevant for IMA architecture.

5.2 Future work

The authors, that applied Metric 3 to the automotive industry used two different views, logic and development view: the former for communication between components and the latter had two purposes. To show the network of the computers and to show deployment of software components in the individual computers. This thesis focused on modelling the system accurately using higraph and thus providing a higraph model view of the system. For future work it should be evaluated if higraph model provides sufficient view for the architects and stakeholders.

It should be investigated whether the weighted dependencies can be integrated into the higraph model and while the suggested weights were relevant in the automotive industry, the aerospace architects and stakeholders need to assess the dependencies and apply weights to suit their industrial needs.

Finally, the higraph and the metrics need to be validated in the industry over longer period of time in guidance of the suggested methods in Chapter 4. It is suggested that implementing the metrics that has the highest probability of succeeding should be chosen first. Metric 3, which has been empirically and theoretically validated, is the obvious choice followed by Metric 2. Metric 1 is suggested to be implemented last due to insufficient maturity. By implementing the other metrics first, more time is given to refine and validate Metric 1.

Bibliography

Aboutaleb, H. and B. Monsuez (2015a). “Handling complexity of a complex system design: paradigm, formalism, and transformations”. *International Scholarly and Scientific Research & Innovation* **9**.

Aboutaleb, H. and B. Monsuez (2015b). “Measuring the complexity of a higraph-based system model: formalism and metrics”. *Procedia Computer Science* **44**.

Aboutaleb, H. and B. Monsuez (2016). “Quantifying system complexity in design phase using higraph-based models”. In: *26th Annual INCOSE International Symposium*. INCOSE.

Albrecht, A. J. and J. E. Gaffney (1983). “Software function, source lines of code, and development effort prediction: a software science validation”. *IEEE Transactions on Software Engineering* **SE-9**.

Aleksa, B. D. and J. P. Carter (1997). “Boeing 777 airplane information management system operational experience”. In: *AIAA/IEEE Digital Avionics Systems Conference. Reflections to the Future*.

Almugrin, S., W. Albattah, O. Alaql, M. Alzahrani, and A. Melton (2014). “Instability and abstractness metrics based on responsibility”. In: *IEEE 38th Annual International Computers, Software and Applications Conference*.

Baruah, S. K., A. Burns, and R. I. Davis (2011). “Response-time analysis for mixed criticality systems”. In: *32nd IEEE Real-Time Systems Symposium*.

Bass, L., P. Clements, and R. Kazman (2012). *Software Architecture in Practise*. 3rd ed. The SEI Series in Software Engineering. Carnegie Mellon.

Bieber, P., F. Boniol, M. Boyer, E. Noulard, and C. Pagetti (2012). “New challenges for future avionic architectures”. *AerospaceLab* **4**.

Briand, L. C., S. Morasca, and V. R. Basili (1996). “Property-based software engineering measurement”. *IEEE Transactions on Software Engineering* **22**.

Bril, R. J., G. Fohler, and W. F. Verhaegh (2008). *Execution times and execution jitter analysis of real-time tasks under fixed-priority pre-emptive scheduling*. Tech. rep. Computer Science, Universiteit Eindhoven.

Broniatowski, D. A. and J. Moses (2014). *Flexibility, Complexity, and Controllability in Large Scale Systems*. Tech. rep. Engineering Systems Division, Massachusetts Institute of Technology.

Butz, H. (2010). *Open Integrated Modular Avionic (IMA): State of the Art and future Development Road Map at Airbus Deutschland*. Tech. rep. Department of Avionics Systems at Airbus Deutschland GmbH.

Card, D. N. and W. W. Agresti (1988). "Measuring software design complexity". *The Journal of Systems and Software* **8**.

Cervantes, H. and R. Kazman (2016). *Designing Software Architectures: A Practical Approach*. The SEI Series in Software Engineering. Carnegie Mellon.

Chandran, S. K., A. Dimov, and S. Punnekkat (2010). "Modeling uncertainties in the estimation of software reliability a pragmatic approach". In: *Fourth International Conference on Secure Software Integration and Reliability Improvement*.

Chen, J. and C. Du (2015). "Schedulability analysis for independent partitions in integrated modular avionics systems". In: *IEEE International Conference on Progress in Informatics and Computing*.

Durisic, D., M. Nilsson, M. Staron, and J. Hansson (2013). "Measuring the impact of changes to the complexity and coupling properties of automotive software systems". *The Journal of Systems and Software* **86**.

Elhag, A. A. M. and R. Mohamad (2014). "Metrics for evaluating the quality of service-oriented design". In: *8th Malaysian Software Engineering Conference*.

Fenton, N. E. and S. L. Pfleeger (1997). *Software Metrics: A Rigorous and Practical Approach*. 2nd ed. International Thomson Computer Press.

Flight Manual A/C JA37.

Gaudan, S., G. Motet, and G. Auriol (2008). "Metrics for object-oriented software reliability assessment - application to a flight manager". In: *Seventh European Dependable Computing Conference*.

Ghasemi, M., S. M. Sharafi, and A. Arman (2015). "Towards an analytical approach to measure modularity in software architecture design". *Journal of Software* **10**.

Gillespie, A. M., M. W. Monaghan, and Y. Chen (2012). "Comparison modeling of system reliability for future nasa projects". In: *Annual Reliability and Maintainability Symposium*. NASA Kennedy Space Center.

Grigg, A. and N. C. Audsley (1999). "Towards a scheduling and timing analysis solution for integrated modular avionic systems". *Microprocessors and Microsystems* **22**.

Grossman, O. and D. Harel (1997). *On the Algorithmics of Higraphs*. Tech. rep. Department of Applied Mathematics and Computer Science.

Gupta, V. and J. K. Chhabra (2009). "Package coupling measurement in object-oriented software". *Journal of Computer Science and technology* **24**.

Harel, D. (1988). “On visual formalisms”. *Communications of the ACM* **31**.

Heinecke, H., K.-P. Schnelle, H. Fennel, J. Bortolazzi, L. Lundh, J. Leflour, J.-L. Mate, K. Nishikawa, and T. Scharnhorst (2004). *AUTomotive Open System ARchitecture - An Industry-Wide Initiative to Manage the Complexity of Emerging Automotive E/E-Architectures*. Tech. rep. AUTOSAR Partnership.

Helfrick, A. (2007). *Principles of Avionics*. 4th ed. Avionics Communications Inc.

Henry, S. and D. Kafura (1981). “Software structure metrics based on information flow”. *IEE Transactions on Software Engineering* **SE-7**.

Jogalekar, P. and M. Woodside (2000). “Evaluating the scalability of distributed systems”. *IEEE Transactions on Parallel and Distributed Systems* **11**.

Kan, S. H. (1995). *Metrics and Models in Software Quality Engineering*. Addison Wesley.

Kim, J.-E., T. Abdelzaher, and L. Sha (2015). “Schedulability bound for integrated modular avionics partitions”. In: *Design, Automation & Test in Europe Conference & Exhibition*.

Klein, M. H., T. Ralya, B. Polak, R. Obenza, and M. G. Harbour (1993). *Practitioner’s Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*. The Kluwer international series in engineering and computer science. Real-time system. Carnegie Mellon.

Kubo, A., H. Washizaki, and Y. Fukazawa (2007). *A Metric for Measuring the Abstraction Level of Design Patterns*. Tech. rep. Department of Computer Science, University of Waseda.

Marinescu, S. O., D. tian TamasSelicean, V. Acretoaie, and P. Pop (2012). “Timing analysis of mixed-criticality hard real-time applications implemented on distributed partitioned architectures”. In: *IEEE 17th Conference on Emerging Technologies & Factory Automation*.

McCabe, T. J. (1976). “A complexity measure”. *IEEE Transactions on Software Engineering* **SE-2**.

Moir, I. and A. G. Seabridge (2006). *Military Avionics Systems*. John Wiley & Sons.

Moir, I., A. Seabridge, and M. Jukes (2013). *Civil Avionics Systems*. 2nd ed. John Wiley & Sons.

Morgan, M. J. (2007). “Boeing 777”. In: Spitzer, C. R. (Ed.). *Digital Avionics Handbook*. 2nd ed. CRC Press. Chap. 28.

Nakamura, T. and V. R. Basili (2005). “Metrics of software architecture changes based on structural distance”. In: *11th IEEE International Software Metrics Symposium*.

Navarro, I., N. Leveson, and K. Lunqvist (2010). “Semantic decoupling: reducing the impact of requirement changes”. *Requirements Engineering* **15**.

Nord, R. L., I. Ozkaya, R. S. Sangwan, and R. J. Koontz (2014). “Architectural dependency analysis to understand rework costs for safety-critical systems”. In: *The 36th International Conference on Software Engineering*.

Pathan, R. M. (2014). “Fault-tolerant and real-time scheduling for mixed-criticality systems”. *Real-Time Syst* **50**.

Pelton, S. L. and K. D. Scarbrough (1997). “Boeing systems engineering experiences from the 777 aims program”. In: *IEEE Transactions on Aerospace and Electronic Systems*. Vol. 33.

Perepletchikov, M., C. Ryan, K. Frampton, and Z. Tari (2007). “Coupling metrics for predicting maintainability in service-oriented designs”. In: *Australian Software Engineering Conference*.

Pillay, R., S. K. Chandran, and S. Punnekkat (2010). “Optimizing resources in real-time scheduling for fault tolerant processors”. In: *1st International Conference on Parallel, Distributed and Grid Computing*.

Prisaznuk, P. J. (2015). “Arinc specication 653, avionics application software standard interface”. In: Spitzer, C. R. et al. (Eds.). *Digital Avionics Handbook*. 3rd ed. CRC Press. Chap. 36.

Qutaish, R. E. A. and A. Abran (2005). *An Analysis of the Design and Definitions of Halstead’s Metrics*. Tech. rep. Ecole de Technologie Superieure, University of Quebec.

Schneidewind, N. F. (1996). “Reliability and risk analysis for software that must be safe”. In: *3rd International Software Metrics Symposium*.

Stewart, D. B. (2006). “Measuring execution time and real-time performance”. In: *Embedded Systems Conference*.

Tomayko, J. E. (2000). *Computers Take Flight: A History of NASA’s Pioneering Digital Fly-By-Wire Project*. NASA.

Troy, D. A. and S. H. Zweben (1981). “Measuring the quality of structured design”. *The Journal of Systems and Software* **2**.

Vestal, S. (2007). “Pre-emptive scheduling of multi-criticality systems with varying degrees of execution time assurance”. In: *28th IEEE International Real-Time Systems Symposium*.

Voss, C. A. and J. Hsuan (2009). “Service architecture and modularity”. *Decision Sciences* **40**.

Yao, J., J. Wu, Q. Li, Z. Xiong, and G. Zhu (2016). “System-level scheduling of mixed-criticality traffics in avionics networks”. *IEEE Access* **4**.

Yau, S. S. and P.-S. Chang (1988). “A metric of modifiability for software maintenance”. In: *The Conference on Software Maintenance*.

Yau, S. S. and J. S. Collofello (1985). “Design stability measures for software maintenance”. *IEEE Transactions on Software Engineering* **SE-11**.

Yourdon, E. and L. L. Constantine (1979). *Structured Design, Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice-hall international.

Zhang, Q. and X. Li (2009). “Complexity metrics for service-oriented systems”. *Second International Symposium on Knowledge Acquisition and Model* **3**.

Appendix

A

Long-list

Table A.1 Long-list that displays authors and general area of their metrics.

Authors	Coupling	Cohesion	Complexity	Abstraction	Availability	Controllability	Dependency Analysis	Flexibility	Maintainability	Modifiability	Modularity	Execution time	Schedulability	Timing Analysis	Response Time Analysis	Resource Analysis	Jitter	Quality	Reliability	Safety	Scalability	Security	Stability
[Yourdon and Constantine, 1979]	✓	✓																					
[Gupta and Chhabra, 2009]	✓																						
[Navarro et al., 2010]	✓																						
[McCabe, 1976]			✓																				
[Qutaish and Abran, 2005]			✓																				
[Henry and Kafura, 1981]			✓																				
[Card and Agresti, 1988]			✓																				
[Aboutaleb and Monsuez, 2016]			✓																				
[Albrecht and Gaffney, 1983]			✓																				
[Zhang and Li, 2009]			✓																				
[Broniatowski and Moses, 2014]			✓			✓		✓															
[Kubo et al., 2007]				✓																			
[Almugrin et al., 2014]				✓																			✓
[Gillespie et al., 2012]					✓				✓										✓				
[Nord et al., 2014]							✓																
[Perepletchikov et al., 2007]									✓														
[Durisic et al., 2013]									✓										✓				✓
[Yau and Chang, 1988]										✓													
[Nakamura and Basili, 2005]										✓													
[Ghasemi et al., 2015]											✓												
[Voss and Hsuan, 2009]											✓												
[Stewart, 2006]												✓	✓	✓									
[Bril et al., 2008]												✓					✓						
[Kim et al., 2015]													✓										
[Grigg and Audsley, 1999]													✓	✓									
[Chen and Du, 2015]													✓										
[Vestal, 2007]													✓										
[Yao et al., 2016]													✓										
[Pathan, 2014]													✓										
[Marinescu et al., 2012]														✓									
[Baruah et al., 2011]															✓								
[Klein et al., 1993]												✓	✓	✓			✓						
[Elhag and Mohamad, 2014]																		✓					
[Gaudan et al., 2008]																			✓				
[Schneidewind, 1996]																			✓				
[Chandran et al., 2010]																			✓				
[Jogalekar and Woodside, 2000]																					✓		
[Yau and Collofello, 1985]																							✓

B

Short-list

Table B.1 Short-list that displays authors and general area of their metrics.

Authors	Coupling	Cohesion	Complexity	Abstraction	Availability	Controllability	Dependency Analysis	Flexibility	Maintainability	Modifiability	Modularity	*Execution time*	*Schedulability*	*Timing Analysis*	*Response Time Analysis*	*Resource Analysis*	*Jitter*	Quality	Reliability	*Safety*	*Scalability*	*Security*	Stability
[Yourdon and Constantine, 1979]	✓	✓																					
[Gupta and Chhabra, 2009]	✓																						
[Navarro et al., 2010]	✓																						
[Henry and Kafura, 1981]			✓																				
[Card and Agresti, 1988]			✓																				
[Aboutaleb and Monsuez, 2016]			✓																				
[Zhang and Li, 2009]			✓																				
[Broniatowski and Moses, 2014]			✓			✓		✓															
[Almugrin et al., 2014]				✓																			✓
[Nord et al., 2014]							✓																
[Perepletchikov et al., 2007]									✓														
[Durisic et al., 2013]									✓										✓				✓
[Yau and Chang, 1988]										✓													
[Ghasemi et al., 2015]											✓												
[Troy and Zweben, 1981]																		✓					
[Elhag and Mohamad, 2014]																		✓					
[Gaudan et al., 2008]																			✓				
[Yau and Collofello, 1985]																							✓

C

Metric 1

Matlab code for Metric 1, structure complexity.

```
function [ structural_complexity ] = higraph(system_matrix , type_matrix ,
     dependency_matrix)
%HIGRAPH Measures the structural complexity given
%system, type, and dependency matrix.

nodes = sum(system_matrix(:))
edges = sum(dependency_matrix(:))
rho = hierarchy(system_matrix)

type_nodes = sum(type_matrix(:))
type_rho = hierarchy(type_matrix)

if edges == 0
    structural_complexity = log2(nodes) + 2*log2(rho) + log2(type_nodes) +
          2*log2(type_rho);
else
    structural_complexity = log2(nodes) + 2*log2(edges) + 2*log2(rho) +
         log2(type_nodes) + 2*log2(type_rho);
end

end
```

D

Metric 2

Matlab code for Metric 2, instability.

```
function [ instability ] = instability( dependency_matrix )
%INSTABILITY Calculates instability given connection and dependency matrix

%D(Pi) Received Dependency
dependency_matrix_normalised = spread_dependency(dependency_matrix);
received_dependency_value = sum(dependency_matrix_normalised ,1);

%R(Pi) Proportional Responsibility
%Dx, total number of the dependent packages in the system excluding the
     sink packages
Dx = ceil(sum(received_dependency_value));
for i = 1:length(received_dependency_value)
    % R(Pi)
    proportional_responsibility(i,:) = (received_dependency_value(i)/Dx);
end

%R'(Pi) Relative Responsibility
% R_max, maximum responsibility value
R_max = max(proportional_responsibility);
for j = 1:length(proportional_responsibility)
     %R'(Pi)

     relative_responsibility(j,:) = (proportional_responsibility(j)/R_max)
          ;
end

% Calculate in- and outgoing connections from nodes
connections_to = transpose(sum(dependency_matrix ,1));
connections_from = sum(dependency_matrix ,2);

instability = zeros(length(dependency_matrix),1);
%I(Pi) Instability
result = 0;
dependency_factor = 0.5;
for m = 1:length(dependency_matrix)
    for n = 1:length(dependency_matrix)
        if (dependency_matrix(m,n) > 0)
            result = result + (1 - (relative_responsibility(n)*
                 dependency_factor));

        end
    end
   if result == 0 && (connections_to(m) + connections_from(m)) == 0
       instability(m,:) = 0;
   else
       instability(m,:) = result / (connections_to(m) + connections_from(m
           ));
       result = 0;
   end
end
end
```

E

Metric 3

Matlab code for Metric 3, complexity and coupling.

```
function [ complex ] = complexity( dependency_matrix )
%COMPLEXITY Calculates the complexity for a given set of
%           data connections represented in a matrix.

connections_to = transpose(sum(dependency_matrix,1));
connections_from = sum(dependency_matrix,2);

complex = 0;
    for i = 1:length(connections_to)
        complex(i,:) = connections_to(i) * connections_from(i);
    end
end


function [ graph, coupling ] = coupling( dependency_matrix, level)
%COUPLING Calculates coupling between partitions of a system
%          given connection matrix and partition set.

size_part = length(level);
coupling = zeros(size_part,1);
incoming = 0;
outgoing = 0;
edges = 0;

for k = 1:(size_part)
    for l = 1:(size_part)
        if k ~= l
            for i = 1:length(level{k})
            from = level{k}(i);
                for j = 1:length(level{l})
                to = level{l}(j);
                outgoing = outgoing + dependency_matrix(from, to);
                incoming = incoming + dependency_matrix(to, from);
                end
            end
            edges = outgoing + incoming;
            graph(k,l) = edges;
            graph(l,k) = edges;
            incoming = 0;
            outgoing = 0;
            edges = 0;
        end
    end
end
coupling = transpose(sum(graph));
```

```
end
```

Lund University **Department of Automatic Control** **Box 118** **SE-221 00 Lund Sweden**	*Document name* MASTER´S THESIS *Date of issue* June 2017 *Document Number* ISRN LUTFD2/TFRT--6036--SE
Author(s) Daniel Jankovic	*Supervisor* Joacim Kämpe, SAAB Aeronautics Pär Hammarström, SAAB Aeronautics Anton Cervin, Dept. of Automatic Control, Lund University, Sweden Karl-Erik Årzén, Dept. of Automatic Control, Lund University, Sweden (examiner) *Sponsoring organization*

Title and subtitle

Metrics for Integrated Modular Avionics Architecture

Abstract

Integrated modular avionics (IMA) architecture is an emerging concept in the military aerospace industry after being successfully implemented in the commercial domain. The highly modular architecture allows multiple aviation applications to execute on the same hardware thanks to defined standards by Aeronautical Radio, Incorporated (ARINC). System architects are responsible for designing and taking advantage of the IMA architecture to meet the requirements set by the stakeholders. They rely much on experience, system knowledge, and design patterns in their work.

This thesis aims to find relevant metrics for system architects when developing IMA architecture in the aerospace industry. A metric survey, with focus on the aerospace and closely related industries, is conducted and broadened to include software and real-time metrics. To find one to three metrics multiple presentations and screenings are held together with a team of domain experts.

Three metrics are selected: structural complexity using Shannon's entropy, instability and abstractness metric, and complexity and coupling metric. The metrics are described in detail and implemented. A small scale system is created to assist and provide better understanding how and what the metrics are measuring. Whether the selected metrics are employable for system architects in aerospace industry still remains to be empirically validated. A proposed validation process is presented for future work.

Keywords

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title 0280-5316		*ISBN*
Language English	*Number of pages* 1-68	*Recipient's notes*
Security classification		

http://www.control.lth.se/publications/