FASSI 2016

The Second International Conference on Fundamentals and Advances in Software Systems Integration

ISBN: 978-1-61208-497-8

July 24 - 28, 2016

Nice, France

FASSI 2016 Editors

Chris Ireland, Open University, UK

Mihaela Iridon, Candea LLC, USA

Fergal Mc Caffery, Dundalk Institute of Technology, Ireland

FASSI 2016

Forward

The Second International Conference on Fundamentals and Advances in Software Systems Integration (FASSI 2016), held between July 24-28, 2016 in Nice, France, continued a series of events started in 2015 and covering research in the field of software system integration.

On the surface, the question of how to integrate two software systems appears to be a technical concern, one that involves addressing issues, such as how to exchange data (Hohpe 2012), and which software systems are responsible for which part of a business process. Furthermore, because we can build interfaces between software systems we might therefore believe that the problems of software integration have been solved. But those responsible for the design of a software system face a number of trade-offs. For example the decoupling of software components is one way to reduce assumptions, such as those about where code is executed and when it is executed (Hohpe 2012). However, decoupling introduces other problems because it leads to an increase in the number of connections and introduces issues of availability, responsiveness and synchronicity of changes (Hohpe 2012).

The objective of this conference is to work toward on understanding of these issues, the trade-offs and the problems of software integration and to explore strategies for dealing with them. We are interested to receive paper from researchers working in the field of software system integration.

We take here the opportunity to warmly thank all the members of the FASSI 2016 technical program committee, as well as the reviewers. We also kindly thank all the authors that dedicated much of their time and effort to contribute to FASSI 2016.

We also gratefully thank the members of the FASSI 2016 organizing committee for their help in handling the logistics and for their work that made this professional meeting a success.

We hope FASSI 2016 was a successful international forum for the exchange of ideas and results between academia and industry and to promote further progress in the area of software systems integration. We also hope that Nice, France provided a pleasant environment during the conference and everyone saved some time enjoy the beautiful French Riviera.

FASSI 2016 Advisory Committee

Chris Ireland, Open University, UK
Mihaela Iridon, Candea LLC, USA
Fergal Mc Caffery, Dundalk Institute of Technology, Ireland
Chris John Lokan, UNSW Canberra, Australia
Christian Percebois, IRIT, Université de Toulouse, France

FASSI 2016

Committee

FASSI 2016 Advisory Committee

Chris Ireland, Open University, UK
Mihaela Iridon, Candea LLC, USA
Fergal Mc Caffery, Dundalk Institute of Technology, Ireland
Chris John Lokan, UNSW Canberra, Australia
Christian Percebois, IRIT, Université de Toulouse, France

FASSI 2016 Technical Program Committee

Hany Ammar, West Virginia University, USA
Marco Autili, University of L'Aquila, Italy
Christian Bird, Microsoft Research, USA
Bara Buhnova, Masaryk University, Czech Republic
Graeme Burnett, Xcordis Fintech, UK
Haipeng Cai, Virginia Tech, USA
Danilo Caivano, University of Bari, Italy
Paul Clarke, Dublin City University / Lero, The Irish Software Research Centre, Ireland
Ip-Shing Fan, Cranfield University, UK
Fabio Fioravanti, University of Chieti-Pescara, Italy
Matthias Galster, University of Canterbury, New Zealand
Anup Gupta, Cognizant Technology Solutions (CTS), UK
Ibrahim Habli, University of York, UK
Alan Hayes, University of Bath, UK
LiGuo Huang, Southern Methodist University, USA
Chris Ireland, Open University, UK
Mihaela Iridon, Candea LLC, USA
Vladimir Itsykson, St. Petersburg State Polytechnic University, Russia
Slinger Jansen, Utrecht University, Netherlands
Foutse Khomh, Ecole Polytechnique de Montréal, Canada
Chris Lokan, UNSW Canberra, Australia
Massimo Marchiori, University of Padua / European Institute for Science, Media and Democracy, Italy
Fergal Mc Caffery, Dundalk Institute of Technology, Ireland
Richard Mordinyi, Vienna University of Technology, Austria
Henry Muccini, University of L'Aquila, Italy
Anh Nguyen Duc, Norwegian University of Science and Technology, Norway
Marc Novakouski, Software Engineering Institute, USA
Tosin Oyetoyan, SINTEF ICT, Trondheim, Norway
Ipek Ozkaya, Carnegie Mellon SEI, USA
Christian Percebois, IRIT, Université de Toulouse, France
Tarmo Ploom, Credit Suisse - Zurich, Switzerland

Dewayne E. Perry, University of Texas at Austin, USA
Patricia Roberts, University of Brighton, UK
Philip Ross, Endava Ltd, London, UK
Massimo Tivoli, Università di L'Aquila, Italy
Gunter Saake, Otto-von-Guericke University Magdeburg, Germany
Mauro Santoro, Università della Svizzera Italiana, Switzerland
Corrado Aaron Visaggio, University of Sannio, Italy
Xiaofei Zhu, L3S Research Center - Leibniz Universität Hannover, Germany

Copyright Information

For your reference, this is the text governing the copyright release for material published by IARIA.

The copyright release is a transfer of publication rights, which allows IARIA and its partners to drive the dissemination of the published material. This allows IARIA to give articles increased visibility via distribution, inclusion in libraries, and arrangements for submission to indexes.

I, the undersigned, declare that the article is original, and that I represent the authors of this article in the copyright release matters. If this work has been done as work-for-hire, I have obtained all necessary clearances to execute a copyright release. I hereby irrevocably transfer exclusive copyright for this material to IARIA. I give IARIA permission or reproduce the work in any media format such as, but not limited to, print, digital, or electronic. I give IARIA permission to distribute the materials without restriction to any institutions or individuals. I give IARIA permission to submit the work for inclusion in article repositories as IARIA sees fit.

I, the undersigned, declare that to the best of my knowledge, the article is does not contain libelous or otherwise unlawful contents or invading the right of privacy or infringing on a proprietary right.

Following the copyright release, any circulated version of the article must bear the copyright notice and any header and footer information that IARIA applies to the published article.

IARIA grants royalty-free permission to the authors to disseminate the work, under the above provisions, for any academic, commercial, or industrial use. IARIA grants royalty-free permission to any individuals or institutions to make the article available electronically, online, or in print.

IARIA acknowledges that rights to any algorithm, process, procedure, apparatus, or articles of manufacture remain with the authors and their employers.

I, the undersigned, understand that IARIA will not be liable, in contract, tort (including, without limitation, negligence), pre-contract or other representations (other than fraudulent misrepresentations) or otherwise in connection with the publication of my work.

Exception to the above is made for work-for-hire performed while employed by the government. In that case, copyright to the material remains with the said government. The rightful owners (authors and government entity) grant unlimited and unrestricted permission to IARIA, IARIA's contractors, and IARIA's partners to further distribute the work.

Table of Contents

(Inter)facing the Business 1
Alexander Hagemann and Gerrit Krepinsky

Automated Infrastructure Management Systems - A Resource Model and RESTful Service Design Proposal to Support and Augment the Specifications of the ISO/IEC 18598/DIS Draft 8
Mihaela Iridon

A Research Roadmap for Test Design in Automated Integration Testing of Vehicular Systems 18
Daniel Flemstrom, Thomas Gustafsson, Avenir Kobetski, and Daniel Sundmark

Case Study: Becoming a Medical Device Software Supplier 24
Kitija Trektere, Fergal McCaffery, Garret Coady, and Matteo Gubellini

A Specific Method for Software Reliability of Digital Controller in NPP 30
Young Jun Lee, Jong Yong Keum, Jang Soo Lee, and Young Kuk Kim

(Inter)facing the Business

(Industry Paper)

Alexander Hagemann

Hamburger Hafen und Logistik AG
Bei St. Annen 1
20457 Hamburg, Germany
Email: hagemann@hhla.de

Gerrit Krepinsky

Hamburger Hafen und Logistik AG
Bei St. Annen 1
20457 Hamburg, Germany
Email: krepinsky@hhla.de

***Abstract*—Over the past decades, a change from singular mainframe applications into complex distributed application landscapes has occured. Consequently, the execution of business processes takes place in a distributed manner, requiring an extensive amount of communication between different applications. It becomes apparent that application interfaces are of overall significance within distributed application landscapes. But in our experience, interfaces usually do not get the required attention during construction, which is in contrast to their importance. Instead, only technical descriptions, e.g., syntactical descriptions, are given and important functional as well as operational aspects have been omitted leading to unstable and unneccesary complex interfaces. To address the aforementioned problems, this paper contributes a comprehensive overview on interface construction. Therefore, all necessary interface specification components, an interface design process and operational migration patterns are given.**

***Keywords*–interface; business process; interface design; interface migration; distributed systems.**

I. INTRODUCTION

In recent years, growing business demands enforced an increasing information technology (IT) support of many business processes. To rule the resulting functional complexity within the IT, several applications are usually necessary. A direct consequence of this fragmentation is the distribution of business processes over applications which have to communicate with each other in order to fulfill the requirements of the business processes. This communication requires well defined interfaces between these applications.

Generally, the design of application interfaces is a difficult and critical task [1], [2], since the behavior of applications belonging to the class of reactive systems, i.e., applications responding continuously to the environment, is determined by their interfaces only [3]. Consequently, badly designed interfaces may lead to functional misbehavior and may propagate internal application problems directly to communication partners [4], [5]. Furthermore, interfaces have relatively long life-cycles and are usually costly to modify. A change of an interface specification always requires either its backward compatibility, or a change of all implementing applications, leading to further problems while launching the new interface into an already running application landscape [6].

Within the literature, a lot of information exists regarding different aspects of interfaces like performance, reliability, routing etc. Typically, these documents either deal with technical protocols only and omit functional interface properties, e.g., the internet protocol [7] or the Blink Protocol [8], or are bound to specific functional domains like the Financial Information eXchange [9] or the FIX Adapted for Streaming [10] protocols. But none of them gives explicit guidelines for an interface design. Other common approaches like service oriented architectures (SOA) [11] or the representational state transfer (REST) [12] represent rather general architectural styles. Both are more suitable giving architectural guidelines for application design, than for the construction of concrete interfaces.

To overcome the above mentioned problems, an approach to construct a consistent interface specification, allowing a regulated communication using stable, understandable and performing interfaces, and its transition into operation will be presented in this paper. Beginning with an overview of all required components to fully specify an interface in Section II, Section III introduces and compares different design approaches for the construction of interface specifications. Finally, Section IV deals with the interface launch into an already running application landscape.

II. INTERFACE BASICS

In order to design an appropriate interface, the general structure of an interface must be considered. Once this has been done, it will become obvious which information must be provided to define an interface.

Drilling down into an interface, which is located in the application layer in the Open Systems Interconnection model (OSI model) [13], typically, a three layered structure, as shown in Figure 1, becomes visible. Each of these layers has a dedicated important purpose that can be summarized as follows:

- *functional layer*: this topmost layer is responsible for the functional semantics of the information exchanged. Using the analog of natural speech, the functional layer defines the meaning of words spoken.
- *protocol layer*: Within this layer the technical protocol used to exchange the information is defined. Similar to natural speech, the protocol layer represents the language spoken, e.g., English.
- *transport layer*: Here, the necessary physical transportation of the information is carried out. This layer

Copyright (c) IARIA, 2016. ISBN: 978-1-61208-497-8

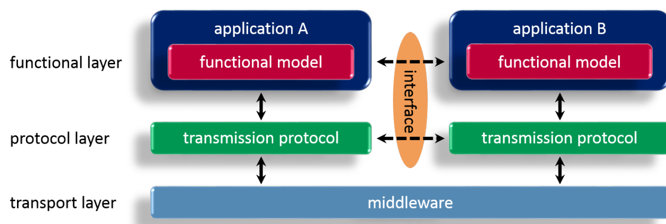

Figure 1. The different layers of an interface. Dotted arrows denote virtual connections within each layer. The communication takes part using the connections denoted by solid arrows.

correlates to the signal transfer using sound waves in a manner similar to natural speech.

Each of these layers communicates logically directly with its counterpart located at the other application. Therefore, a layer physically passes the information to its underlying layers until the information is physically transported to the other application. At this point, the information is passed upwards up to the corresponding layer. Only if both sides within one layer use identical functional models or transmission protocols, respectively, communication will take place. Otherwise, the communication is broken.

Given the layered structure of an interface, different aspects arise which must be considered during the design, implementation, integration and operational phases of an interface lifecycle. These aspects focus on different issues and enable the development of robust interfaces. All aspects are independent with respect to each other, focusing on a specific property an interface must satisfy.

A. Functional aspect

While two or more applications are communicating with each other over an interface, the applications assume different functional roles, called *server* and *client*, respectively.

An application is called *server* with respect to an interface if it is responsible for the business objects, business events and related business functions that are exposed to other applications through this interface. If business objects and business functions of different business processes are affected, the necessary access messages may be combined into a single interface.

Providing an interface is equivalent to defining an interface contract [6] that must be signed by an application in order to communicate with the *server*. The interface may support synchronous and asynchronous communication as well as message flows in both directions, i.e., sending and receiving messages.

Note that this definition deviates slightly from the commonly used client-server definition where the server offers a service which can be accessed by clients via a synchronous request-reply communication protocol only [14]. Because synchronous communications couples server and client tightly at runtime, asynchronously based communication should be preferred, avoiding these disadvantages [15].

A *client* is an application consuming an interface provided by a server. Despite the fact that the interface contract is initially defined by the *server*, a common agreement on the contract is made when the *client* connects to the server. Thereafter, none of the participating applications may change the interface contract without agreement of the other party.

Often an application assumes multiple roles with respect to different interfaces concurrently, i.e., the application can be *server* and *client* simultaneously. It is important to emphasize that this behavior is valid with respect to different interfaces only while for a single interface, the roles of the participating applications are always unambiguous.

B. Semantical aspect

The semantical aspect focuses on the kind of information that may be exposed by the *server* via an interface. Generally, any internal implementation detail of the *server*, i.e., the server model, must never be exposed on an interface. Instead, the information exposed must always be tied to the underlying business processes, thus binding the interface implementation to the domain model [6], [16].

Integration within an IT application landscape requires the decoupling of business and software design due to different responsibilities. In other words the business model and the software model usually have different life cycles which must be decoupled to reduce the dependencies between business and software developers. Therefore, an integration model, linked to the domain model, should be used on interfaces thus binding their implementation to the domain model [16]. The integration model finally conceals all internal application models and details.

An interface itself consists of a set of messages, containing *business objects* or *business events* only [17]. This set of exposed information is naturally restricted due to the responsibility of the *server*, i.e., only the *business objects* or *business events* the *server* is responsible for may be communicated via the interface.

C. Dynamical aspect

An important aspect of an interface is its dynamical behavior describing all valid message sequences on the interface. Since all messages received are processed within a specific context inside the application, there exist important constraints with respect to the message sequence. Thus, a message received out of sequence will not be processed by the application, instead this will result in an error.

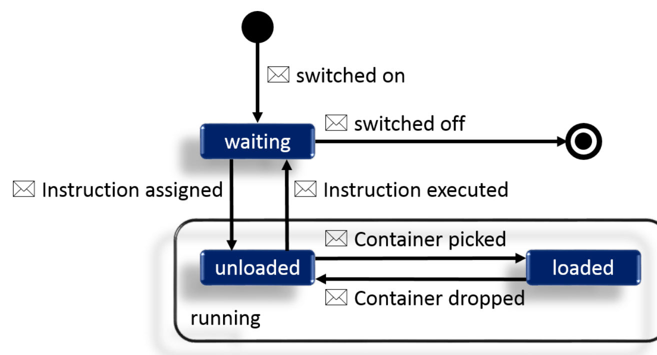

Figure 2. Example of a simplified state machine describing the dynamical behavior of an interface.

Consequently, the dynamical behavior must be described using an appropriate description. Using sequence diagrams of the Unified Modelling Language (UML) is not sufficient for this case, since they describe specific communication examples only. Especially runtime problems, e.g., race conditions, can not be described holistically using sequence diagrams. Instead, it is strongly recommended to use finite state machines which allow a complete description of the dynamical behavior, see Figure 2 for an example.

D. Operational aspect

Usually, each interface requires the usage of specific infrastructure depending on the used transmission protocol, e.g., a web server in case of REST over Hypertext Transfer Protocol (HTTP) or a Java Messaging Service (JMS) server. To ensure the correct usage of an interface the required infrastructure, its deployment and the message channel topology must be defined. The latter one defines the communication structure, i.e., broadcast or point-to-point communication [15].

E. Interface components

Given the different aspects presented so far, each of them describing a different important issue with respect to interfaces, the necessary components for a complete interface specification can be derived:

- *message description*: Syntactical descriptions of all messages exchanged over the interface.
- *dynamic description*: The dynamic behavior of the interface must be fully specified. This specification includes all possible message sequences and the behavior of the applications in case of errors.
- *semantic description*: The meaning of messages on the interface must be specified, i.e., their functional behavior within the comprehensive business process. This description must include the meaning of all individual message fields.
- *infrastructure description*: A description of the necessary infrastructure must be provided.
- *quantity description*: The non-functional performance requirements for the interface must be described.

It is important to notice that an interface specification is a signed bilateral contract, which may be changed by mutual agreement of all participating parties only. This contract is represented by the set of artifacts as described above, so none of the artifacts given there may be missed.

III. Interface design styles

The main problem to be solved in interface design concerns the intended functional semantic on the interface. It directly influences the kind of service offered by the *server* and therefore the necessary number and style of all messages.

Looking at existing interfaces, they can be categorized to our experience by their semantic design styles: CRUD based interfaces, use case based interfaces and business process based interfaces, each of them described in detail in the following sections.

A. CRUD based design

The Create, Read, Update, Delete (CRUD) based design directly uses the business objects described within the requirements and ignores any given business context. This results in interfaces consisting of a minimal set of messages, representing a set of CRUD messages for every business object the *server* is functionally responsible for. Besides the advantages of requiring very little design efforts and being very stable, this interface design style has some important disadvantages.

First, the interface bears absolutely no business context, leading to severe difficulties in understanding the underlying business processes [1]. Second, the read operation demands synchronous communication which represents an explicit control flow leading to a tight coupling of applications [4], [17] and third, missing business context either leads to a distribution of business functions over the *clients* or to business objects incorporating the results of applied business functions.

B. Use case based design

An interface design based on use cases rests upon requirements formulated from the perspective of the primary actors for individual systems only [18], i.e., the underlying business process is not directly present. Due to the characteristics of use cases, describing non-interrupted interactions with the system [19] that represent the view of the primary actor [18], these requirements are limited to the context of single activities which are typically independent with respect to each other. A representation of the underlying business process triggering the desired activities is missing and therefore difficult to reconstruct.

These preconditions usually lead to rather fine granular and use case oriented interfaces comprising of a large number of messages, carrying specific use case based information only. Some important consequences arise from this design style. First, all business functions that are identical from the business process point of view, are hard to identify based on a use case analysis only. The absence of a business context leads to an interface design supporting individual use cases, which bear no evident business process semantics. Consequently, these interfaces usually offer a broad range of identical functionalities named differently. Second, the missing business context significantly increases the difficulty to understand the functional behavior of the interface over time [1], leading to serious problems in its usage. As a consequence, further unnecessary messages are often introduced in order to provide some use case specific information. Third, the lower level of abstraction of a use case - compared to the business process - leads to a rather fine granular interface structure. Performance issues may arise with this interface style due to the enforced frequent interface access [15]. And fourth, synchronous communication often arises in order to collect all necessary information to execute the use case, so a control flow arises [17] leading, again, to a tight coupling of applications [4].

Note, that using a use case based design must not lead compulsorily to a bad interface design. But given the size of current applications with their numerous use cases and the typical usage of distributed programming teams within industrial projects, the necessary refactoring to introduce an appropriate abstraction on the interface is usually omitted in our experience.

C. Business process based design

This design uses business process descriptions and further requirements formulated with respect to those descriptions, to align between individual business process activities and applications. Using the Business Process Model and Notation (BPMN) and representing applications via pools, interfaces can be directly derived from the exchanged information between individual business activities within the pools.

The resulting interfaces focus on business semantics and directly support objects, events and functions of the business processes, thus leading to a business model directly bound to the interfaces [16] with following consequences.

Business processes support a high level of abstraction, thus leading to rather coarse granular interfaces with respect to the number of messages. The communication is driven by business events, so asynchronous communication is naturally supported, leading to data flows [17]. Finally, the functionality provided by the server within the business processes becomes rather clear, i.e., the business context is represented on the interface.

D. Design example

To explain and clarify the differences between these design styles, the simplified process of loading a truck at a container terminal will serve as an example throughout this section. This process consists of the following steps, executed in the given order:

- *order clearance*: the customer gives an order to the container terminal to load a container on a truck.
- *load clearance*: in order to deliver the container, several clearances must be given, e.g., by customs and the container owner.
- *transport planning*: the container terminal plans the necessary equipment to execute the order.
- *load container*: the container is loaded on the truck using the planned equipment.

Two applications shall be constructed in order to implement the process: the *Administration*, dealing with the administrative parts of the process, and *Operating*, handling the physical transport of the container. An interface between both applications will be designed according to the design style considered, thus showing the differences between the design approaches.

1) CRUD based design: All relevant business objects of the truck loading process are represented as classes which have methods to create, read, update and delete the object. These methods represent the interface of the owning application, i.e., the *server*, and are called by the *clients*, in order to execute the business process.

For example, after creating an order using `createOrder()`, the *Administration* calls `createInstruction()` to start the loading of the container on a truck. Subsequently, *Operating* calls `readCustomsClearance()` and `readReleaseOrder()` to check if the container is released to be loaded on a truck. The corresponding return objects must be interpreted within *Operating* to make this decision. If the container has been loaded, *Operating* finally calls `deleteOrder()`, `deleteCustomsClearing()` and `deleteReleaseOrder()` to clear the *Administration*.

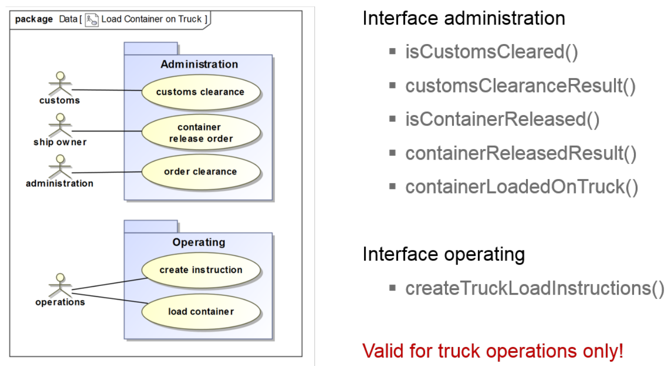

Figure 3. Constructed interface (right) resulting from applying the use case based design approach.

It becomes clear that both applications, i.e., *Administration* and *Operating* must implement some part of the underlying business logic to deal with these type of interfaces. Since the interface style bears no business semantics, the underlying business process cannot be reconstructed easily. Note that the size of the interface directly depends on the number of business objects the server is responsible for.

2) Use case based design: Based on the requirements of the truck loading process, corresponding use cases like order clearance or create instruction can be derived, as shown in Figure 3. Each of these use cases handles a specific functional aspect with respect to its primary actor. The underlying business process is executed through a set of use cases interacting with each other.

For example, if an order has been given, *Administration* calls `createTruckLoadInstruction()` to initiate the container transport. Prior to loading, *Operating* checks the container release status, using `isCustomsCleared()` and `isContainerReleased()`. If the container has been released, it is loaded on truck and *Operating* informs *Administration* via `containerLoadedOnTruck()` that the order has been executed. *Administration* may then clean up its internal data structures.

As depicted on the right side of Figure 3, the resulting interface contains a lot of methods for specific actions, i.e., the level of abstraction is rather low. Consequently the interface is valid for truck operations only and would require a couple of additional methods to incorporate e.g., vessel and train operations.

Furthermore the interface introduces synchronous communication, as indicated by, e.g., the method pairs `isCustomsCleared()` and `customsClearanceResult()`, leading to a blocking of *Operating* while accessing the information.

3) Business process based design: In this case, the business process itself serves as basis for interface design. Using BPMN, the process of truck loading can be mapped onto the applications as shown on the left side of Figure 4. Due to the given high level of abstraction within the business process, it is valid for all types of carriers, i.e., no further messages are necessary to include vessel and train operations.

Once an order has been given, *Administration* informs *Operating* via `orderPlaced()` that a new order has been

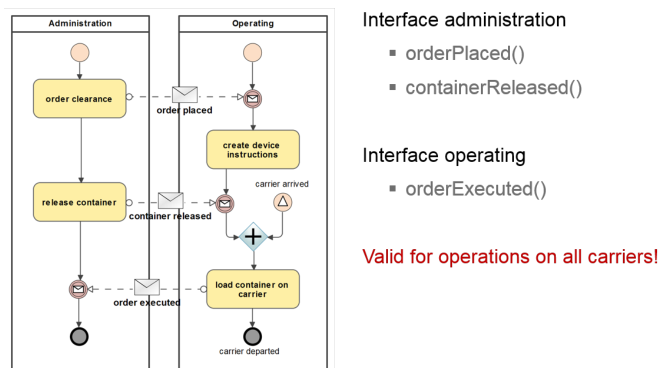

Figure 4. Constructed interface (right) resulting from applying the business process based design approach.

accepted. Within *Operating*, all necessary instructions for container loading will be created. Once the truck has arrived and *Administration* has published via `containerReleased()` that the container is released to be loaded on a truck, the physical moves are executed. Afterwards, `orderExecuted()` informs *Administration*, to clean up its internal data structures.

The dynamical behavior of the interface can be derived directly from the BPMN description, see left side of Figure 4. The resulting interface is quite small, meaningful and abstract, so other carriers can easily be included. Additionally, the communication between both applications is asynchronous. Note that both applications, *Administration* and *Operating*, do not technically depend on each other, instead they simply publish their information without knowing the receiver, resulting in a data flow [17].

E. Comparison

To give a recommendation for a specific interface design style all design approaches described above have been compared to each other using typical interface design goals like robustness, performance and understandability [1].

1) Robustness: Interfaces are crucial with respect to the stability of the overall application landscape. Poorly designed interfaces may propagate internal application errors during runtime, thus causing damage within other applications [4], [1]. Robustness is achieved by avoiding functional distribution, distributed transactions [5] and semantical ambiguity.

In case of a CRUD interface, the information provided by the interface must be functionally interpreted by the client since the *server* informs about changes on business objects only without any functional context. This leads to multiple and distributed implementations of business functions according to the usage of the interface. In contrast, the use case and business process based design styles can both concentrate the business functions within the *server*, so no functional distribution will arise.

In general, distributed technical transactions can be avoided in all three design approaches. But modelling a control flow instead of a data flow bears a higher risk of introducing distributed transactions within the application landscape, due to the usage of synchronous communication.

None of the design approaches specifically supports the construction of an efficient message field structure nor prohibits the introduction of content based constraints.

2) Performance: Obviously, interfaces must satisfy the required performance, i.e., they must be able to deal with the given quantity description. Otherwise, the business process will not work correctly since required business functions may not be executed in time. Performance is supported by designing minimal interfaces with respect to the number of messages and avoiding synchronous communication [3], [15].

The more abstract the interface is, the less messages are needed due to the restriction of transmitting core concepts only. With a CRUD based design, the most abstract design is chosen while a use case based design includes relatively less functional abstraction.

Asynchronous communication is usually directly supported in the business process based design, while the other two approaches support a rather synchronous communication style. This holds especially for the CRUD based design, where the `read()` operation always enforces synchronous communication.

3) Understandability: Well designed interfaces must have a strong and documented relation to the underlying business context [1] thus ensuring a good usability of the interface. This will enhance the cost efficiency of the interface over time since a much better acceptance of the interface within the development teams will arise because the interface will be easier to learn, remember and use correctly [1].

Understandability is given by a strong functional binding between the business model and the implementation [16], the usage of business objects and business events as message content [3], [15] and a meaningful message naming schema.

Naturally, a business process based design leads to a direct mapping between interface and business process description thus enriching the interface with a comprehensive business context. On the contrary, a CRUD based design bears no business context at all due to its high level of abstraction.

Although all three approaches directly support the exchange of business objects, differences occur considering the publication of business events. A CRUD based design supports none of them per se, i.e., this approach forces a mapping of business events onto business objects. This will lead to serious problems in understanding the dynamical behavior of the application landscape. Using a business process based design instead, the published business events can be directly derived from the underlying business process. In contrast, a use case based design does not primarily focus on business events but on individual user operations thus obscuring the business context.

While the business process and the use case based designs both support message naming schemas providing a rich functional context, a CRUD based design uses only the given names for create, read, update and delete messages.

4) Recommendation: Considering the above mentioned desgin goals and the important advantage of supporting a direct binding between business model and interface design, the business process based design is the recommendated design style for interfaces, leading to the best design compromise.

Copyright (c) IARIA, 2016. ISBN: 978-1-61208-497-8

IV. Interface Operations

Complex application landscapes require the rollout of interface changes without shutting down all applications. To achieve this goal interfaces must be versioned and deployed during runtime, using the migration patterns described below.

A. *Interface versioning*

Every interface specification evolves over time due to syntactic, semantic or dynamic changes on the interface. These changes lead to different versions of the interface specification which are not compatible to each other. Therefore, the implementing applications must implement the correct version of the interface specification. In a complex application landscape, this is a common situation [6].

In order to guarantee a unique identification of a specific interface occurrence over time, each individual interface occurrence must have a version number [6]. Any change on an interface leads to a new interface version [6]. This includes syntactical changes in any message, changes within the message sequence flow, i.e., all changes of the dynamic behavior, and changes of the semantic behavior. Even the obviously simple cases of adding either a field to an existing message or introducing a new message to an interface represents a semantical change of the interface. This requires compatibility of the receiving application with the new interface specification version. Otherwise, severe problems may arise, if, e.g., a client executes syntactical message checks based on a specific interface version.

B. *Big bang migration pattern*

The simplest approach of an interface migration is *big bang*, where all applications are shutdown, redeployed and restarted at the same time, resulting in

$$1 + c \tag{1}$$

migration steps, where c denotes the number of participating clients. In case of a fallback, the server and all clients must be redeployed again.

C. *Client first migration pattern*

Within this pattern, the migration path is dominated by the clients. Each client will be successively migrated onto a new version that can handle both interface versions in parallel, as shown in Figure 5. In steps 1 and 2, the clients are changed to support additionally the new interface specification version. In step 3, the server is merged to the new interface implementation. Steps 4 and 5 are necessary to remove the support of the previous interface specification version from the clients.

After finishing all client migrations, the server will be upgraded to support the new interface version. Afterwards, all clients will be updated a second time in order to remove the support of the old interface version. During steps one to four of this migration path, the server will receive messages with a wrong interface version that must be ignored by the *server*.

The *client first migration* pattern will result in

$$1 + 2 * c \tag{2}$$

deployments, where c denotes the number of clients connected to the server. An advantage of this migration path is that

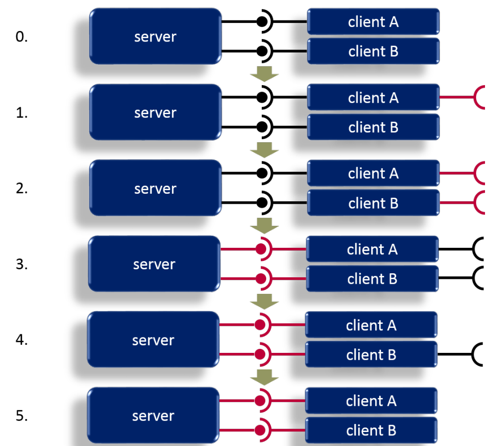

Figure 5. Steps of the client first migration pattern. The new interface version is denoted red.

clients can be upgraded independently from each other, i.e., no temporal coupling of the individual client migrations exist.

The price for this migration behavior is the necessary number of deployments : each client must be deployed two times, while the server is deployed only once. Furthermore, in case of a failure, the operational safe position of step 2 must be reached again. This is done by falling back with the server supporting the old interface version only and all clients whose support of the old interface version has been removed so far, requiring

$$1 + c_+ \tag{3}$$

steps, where c_+ denotes the number of clients migrated after the server migration.

D. *Server first migration pattern*

In contrast to the *client first migration* approach, the migration path can be reversed resulting in a server migration first followed by client migrations, see Figure 6. At step 1, the server provides support for two interface specification versions. In steps 2 and 3, both clients are merged successively. Finally, support of the previous interface specification version is removed from the server implementation, resulting in

$$2 + c \tag{4}$$

deployments. Again c denotes the number of participating clients. The advantage of this pattern is, that the number of deployments is

$$(1 + 2 * c) - (2 + c) = c - 1 \tag{5}$$

less than with the *client first migration* pattern. Note that during steps one to three of the migration path both *clients* A and B will receive invalid messages, which must be ignored, due to the concurrent interface version support of the server.

If a failure on the interface occurs within the migration path, all clients upgraded so far must fall back onto the previous interface version using c_+ rollout steps, where, again, c_+ denotes the number of clients migrated after the server migration. Thus, the operational safe position of step 1 is reached again.

Copyright (c) IARIA, 2016. ISBN: 978-1-61208-497-8

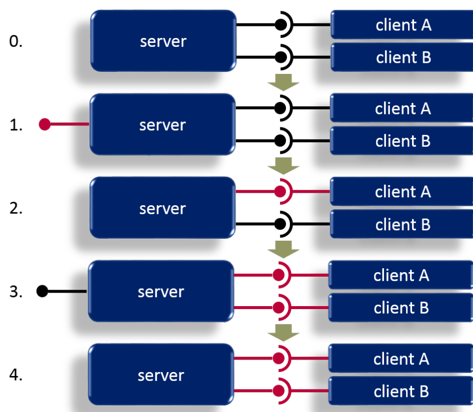

Figure 6. Steps of the server first migration pattern.

E. Comparison

The main differences between the migration patterns are the number of rollout and fallback steps and the required support of multiple interface versions within the applications. Beside the advantages of a lacking necessity to support multiple interface versions and a minimal number of rollout steps, the *big bang* pattern bears a high risk during fallback situations where multiple applications must fallback in parallel. Therefore, this pattern is only recommended if the number of clients is very small and a simultaneous fallback is organizational manageable.

Considering the other strategies, both migration patterns reduce the risk involved with a possible fallback compared to *big bang* at the cost of some additional rollout steps. Since the *server first migration* pattern requieres less rollout and fallback steps than the *client first migration* pattern, it is the recommended rollout strategy.

V. SUMMARY

Due to the growing distribution of business functionality, interfaces have became very important for the behavior of an application landscape. Badly designed interfaces have a critical impact on the functional and operational behavior. To overcome these problems, this paper presented a structured and holistic approach of handling interfaces during design, build and runtime as follows.

Interfaces serve as contracts between applications. Thus it is inevitable to define the artifacts *message description*, *dynamic description*, *semantic description*, *infrastructure description* and *quantity description* to properly describe an interface with respect to the different aspects. In order to construct an interface, different design approaches have been presented and compared to each other. It turns out that the *business process based design* approach is most likely leading to the best result with respect to robustness, performance and understandability. Finally, different migration patterns have been presented introducing a new interface version into production environment. Due to the minimal number of required fallback steps in case of a severe error and one additional rollout step compared to the *big bang* pattern the *server first migration* pattern is recommended, at least for larger application landscapes.

ACKNOWLEDGMENT

The authors would like to thank their colleague Christian Wolf for valuable comments.

REFERENCES

[1] M. Henning, “API Design Matters,” ACM Queue Magazine, vol. 5, 2007.

[2] J. Bloch, “How to Design a Good API and Why it Matters,” 2006, URL: http://landawn.com/How to Design a Good API and Why it Matters.pdf [accessed: 2016-06-08].

[3] R. J. Wieringa, Design Methods for Reactive Systems. Morgan Kaufmann Publishers, 2003, ISBN: 1-55860-755-2.

[4] M. Nygard, Release It!: Design and Deploy Production-Ready Software. O’Reilly, Apr. 2007, ISBN: 978-0978739218.

[5] U. Friedrichsen, “Patterns of Resilience,” 2016, URL: http://de.slideshare.net/ufried/patterns-of-resilience [accessed: 2016-06-06].

[6] B. Bonati, F. Furrer, and S. Murer, Managed Evolution. Springer Verlag, 2011, ISBN: 978-3-642-01632-5.

[7] S. Deering and R. Hinden, “Internet Protocol, Version 6 (IPv6) Specification,” The Internet Society, Specification, 1998.

[8] “Blink Protocol,” 2012, URL: http://blinkprotocol.org/ [accessed: 2016-06-06].

[9] “Financial Information eXchange,” 2016, URL: https://en.wikipedia.org/wiki/Financial_Information_eXchange [accessed: 2016-06-06].

[10] “FAST protocol,” 2016, URL: https://en.wikipedia.org/wiki/FAST_protocol [accessed: 2016-06-06].

[11] “Service-oriented architecture,” 2016, URL: https://en.wikipedia.org/wiki/Service-oriented_architecture [accessed: 2016-06-08].

[12] R. Fielding, “Architectural Styles and the Design of Network-based Software Architectures,” dissertation, University of California, Irvine, 2000.

[13] H. Kerner, Rechnernetze nach ISO-OSI, CCITT. H. Kerner, 1989, ISBN: 3-900934-10-X.

[14] “Client-server model,” 2016, URL: https://en.wikipedia.org/wiki/Client-server_model [accessed: 2016-03-03].

[15] G. Hohpe and B. Woolf, Enterprise Integration Patterns. Addison-Wesley, 2012, ISBN: 978-0-133-06510-7.

[16] E. Evans, Domain Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley, 2004, ISBN: 0-321-12521-5.

[17] R. Westphal, “Radikale Objektorientierung - Teil 1: Messaging als Programmiermodell,” OBJEKTspektrum, vol. 1/2015, 2015, pp. 63–69.

[18] A. Cockburn, Writing Effective Use Cases. Addison-Wesley, 2001, ISBN: 978-0-201-70225-5.

[19] B. Oestereich, Objektorientierte Softwareentwicklung: Analyse und Design mit der UML 2.0. Oldenbourg, 2004, ISBN: 978-3486272666.

Copyright (c) IARIA, 2016. ISBN: 978-1-61208-497-8

Automated Infrastructure Management Systems

A Resource Model and RESTful Service Design Proposal to Support and Augment the Specifications of the ISO/IEC 18598/DIS Draft

Mihaela Iridon
Cândea LLC for CommScope, Inc.
Dallas, TX, USA
e-mail: iridon.mihaela@gmail.com

***Abstract*— Automated Infrastructure Management (AIM) systems are enterprise systems that provision a large number and variety of network infrastructure resources, including premises, organizational entities, and most importantly, all the telecommunication and connectivity assets that enable network infrastructure to operate locally and across vast geographical areas. The representation of infrastructure elements managed by such systems has never been normalized before, making integration – a challenging undertaking on its own – an even more difficult task, requiring specialized knowledge about the systems and the infrastructure data they provision. Such details are most relevant given the complexity and variety of telecommunication infrastructure systems and the widespread need for external or custom applications to gain access to the data and features built in to these AIM systems. This year however, the international standards organization is scheduled to release new standard ISO/IEC 18598 that will provide standardization and sensible guidelines for exposing data and features of AIM systems and thus to facilitate the integration with custom clients for these systems. CommScope, an active contributor in defining these standards, has implemented to a large extent these specifications for their imVision system and in doing so, decided to capture some relevant details that would bring more clarity, add context, and provide further guidelines to the information described by the standards document. In order to achieve these goals and in an attempt to lead the way towards a robust AIM system design that aligns with these standards, this paper elaborates on the recommended models. It also intends to share architectural and technology-specific considerations, challenges, and solutions adopted for the CommScope's imVision standards-based API, so that they may be translated and implemented by other organizations that intend to build - or integrate with - an AIM system in general.**

Keywords-automated infrastructure management (AIM); system modeling; ISO/IEC 18598.

I. Introduction

ISO/IEC have recently put forth a set of requirements and guidelines for modeling and provisioning Automated Infrastructure Management (AIM) systems [1] that will help consolidate how such systems represent the assets and entities they provision, as well as enable custom integration solutions with these systems. Identifying and organizing AIM system's assets in a logical and structured fashion allows for an efficient access and management of all the resources administered by the system.

As with every software system and more so with enterprise-level applications, domain modeling is of crucial importance as it helps define, refine, and understand the business domain, facilitating the translation of requirements into a suitable design [5]. However, special-purpose models can and should be designed for various layers of a system's architecture [11]. When a system exposes integration points to outside agents or clients, it is imperative to define clean boundaries between the system's domain and the integration models [6] [4]. Stability of integration models is just as important as versioning for extensible systems, while allowing the domain models, structural or behavioral, to evolve independently of all other models that the system relies on [2] [3].

The first half of this paper (Section II) will present the relevant resource models from the perspective of a RESTful services design [2] [10] [13], with focus on the underpinning structures and the telecommunication assets, as proposed and used by CommScope's imVision API. This section also presents a solution for handling a large variety of hardware devices while avoiding a large number of URIs for accessing these resources. Section III discusses system architecture, patterns and design-specific details. Section IV presents some of the challenges encountered during the realization of the system design, solutions employed, and finally joining all the discussion points to a conclusion in Section V.

II. AIM System Domain Analysis and Resource Modeling

The resource model presented in this paper employs various design and implementation paradigms. However, the only types exposed by the system, i.e., all concrete resource types, can be viewed and modeled as simple POCOs (Plain Old CLR Objects for the .NET platform) or POJOs (Plain Old Java Objects for the Java EE platform). These models represent merely data containers that do not include any behavior whatsoever. Such features are specific to the physical entities being modeled and are highly customized for a given system. The model proposed here serves the purpose of defining a common understanding of the data that can be exchanged with an AIM system while any specific

behavior around these data elements is left to the implementation details of the particular AIM system itself.

As opposed to stateful services design principles (such as SOAP and XM -RPC-based web services) - where functional features and processes take center stage while data contracts are just means to help model those processes [4] [11], in RESTful services the spotlight is distinctly set on the transport protocol and entities that characterize the business domain. These two elements follow the specifications of Level 0 and 1, respectively, of the RESTful maturity model [7] [13]. The resources modeled by a given system also define the service endpoints (or URIs), while the operations exposed by these services are simple, few, and standardized (i.e. the HTTP verbs required by Level 2: GET, POST, PUT, DELETE, etc.) [10] [13]. Nonetheless, in both cases, a sound design principle (as with any software design activity in general) is to remain technology-agnostic [5] [6] [11].

A. Resource Categories Overview and Classification

The entities proposed in the Standards document [1] are categorized by the sub-domain that they are describing as well as their composability features. At the high-granularity end of the spectrum we will find entities that deal with the location of networking centers (sites, cities, buildings, rooms, etc.) while at the other end of the spectrum we have the smallest assets that the system manages (modules and ports, outlets and cables). This classification helps define a model that aligns well with the concept of separation of concerns (SoC), allowing common features among similar entities to be shared effectively, with increased testability and reliability.

The Standards document proposes the following categories of resources to be provisioned by an AIM System, as shown in Table 1.

TABLE I. RESOURCE CATEGORIES AND EXAMPLES OF CONCRETE TYPES

PREMISES	Geographic Area, Zone, Campus, Building, Floor, Room
CONTAINERS	Cabinets, Racks, Frames
TELECOM ASSETS	Closures, Network Devices, Patch Panels, Modules, Ports, Cables, Cords
CONNECTIVITY ASSETS	Circuits, Connections
ORGANIZATIONAL	Organization, Cost Center, Department, Team, Person
NOTIFICATIONS	Event, Alarm
ACTIVITIES	Work Order, Work Order Task

Some elements listed above may not be relevant to all AIM systems. The Standards document intends to capture and categorize all elements that could be modeled by such a system. It also suggests a common terminology for these categories so that from an integration perspective there is no ambiguity in terms of what these assets or entities represent and what their purpose is. Otherwise stated, it defines at high-level the ubiquitous integration language by providing a clear description and classification of the main elements of an AIM system. This paper takes these recommendations, materializes them into actual design artifacts, and proposes a general-purpose layered architecture for the RESTful AIM API system.

B. Common Abstraction Models

Since all resources share some basic properties, such as name, identifier, description, category, actual type (that identifies the physical hardware components associated with this resource instance), and parent ID, it is a natural choice to model these common details via basic inheritance, as shown in Figure 1. In order to support a variety of resource identifiers (i.e., Globally Unique Identifier, integer, string, etc.) the `ResourceBase` class is modeled as a generic type, with the resource and parent identifier values of generic `TId`.

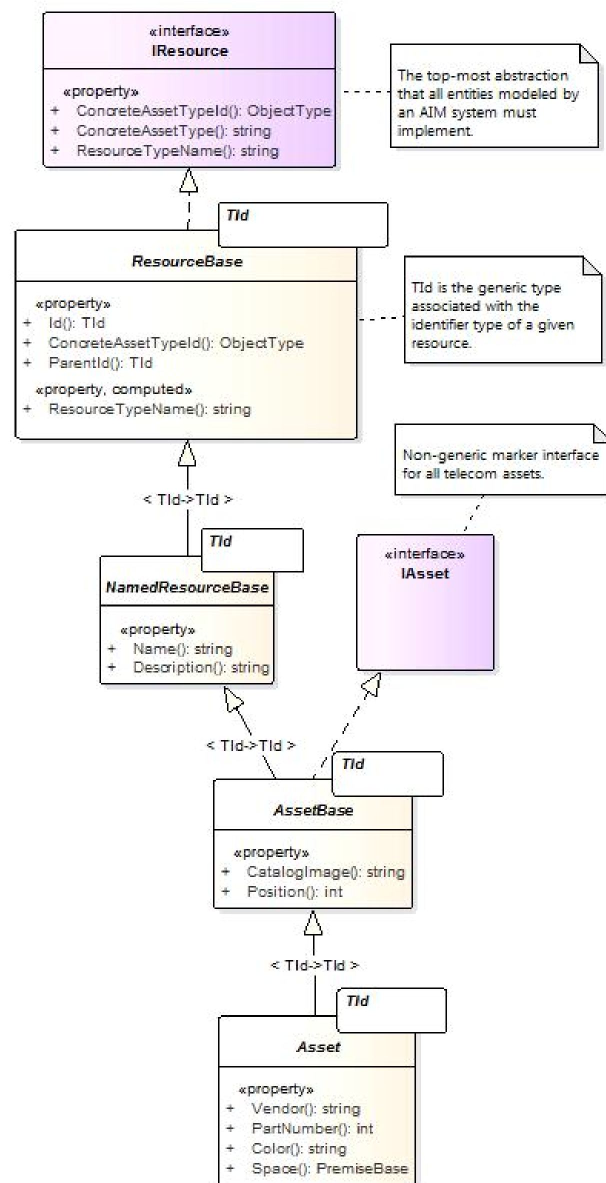

Figure 1. Resource Base Models

Of particular interest are telecommunication assets – the core entities in AIM systems – a class of resource types, which all realize the `IAsset` interface, an abstraction used as a marker on the type. These entities will be presented in the next sub-section.

C. Resource Model Design

1) Premise Elements

Company's network infrastructure can be geographically distributed across multiple cities, campuses, and/or buildings, while being grouped under one or more sites – logical containers for everything that could host any type of infrastructure element. At the top of the infrastructure-modeling hierarchy, there are the premises, which model location at various degrees of detail: from geographic areas and campuses to floors and rooms. Composition rules or restrictions for these elements may be modeled via generic type constraints, unless these rules are not enforced by a given system. Figure 2 shows the standards-defined premise entities, their primary properties, and the relationships between them.

2) Telecom Connectivity Elements

The main assets of a network infrastructure are its telecommunication resources, from container elements, such as racks and cabinets, to switches and servers, network devices (e.g. computers, phones, printers, cameras, etc.), patch panels, modules, ports, and circuits that connect ports via cables and cords. The diagram included in Figure 3 shows these asset categories modeled via inheritance, with all assets realizing the **IAsset** marker interface. As is the case for CommScope's imVision system, the type of the unique identifier for all resources is an integer; hence, all resource data types will be closing the generic type **TId** of the base class to **int**: **ResourceBase<int>**. This way, the RESTful API will expose these AIM Standards-compliant data types in a technology- and implementation-agnostic way that reflects the actual structure of the elements, while generics and inheritance remain transparent to integrators, regardless of the serialization format used (JSON, XML, SOAP). This fact is illustrated in Figure 5, which shows a sample rack object serialized using JSON.

In addition to the elements shown in Figure 3 that support a persistent representation of the data center's telecom assets, there are those that enable circuits to be specified: cables, connectors, and cords. They play a role in defining the connectivity dynamics of the system. Figure 4 shows the primary resources for modeling this aspect of an AIM system.

3) Organizational Elements

Some AIM systems may desire to provision entities that describe the organization responsible for maintaining and administering the networking infrastructure. For example, tasks around the management of connectivity between panels and modules is usually represented by work orders that comprise one or more work order tasks. Such tasks are then assigned to technicians, which report to a manager, which in turn belongs to a department, and so on. The model for these elements is not included here as it is straightforward but is available upon request.

4) System Notifications and Human Activity Elements

Hardware components of AIM systems, e.g., controllers, discoverable/intelligent patch panels and in some instances intelligent cords (e.g. CommScope's Quareo system) allow continuous synchronization of the hardware state with the logical representation of the hardware components.

This synchronization is facilitated by the concept of events and alarms that are first generated by controllers (alarms) and then sent for processing by the management

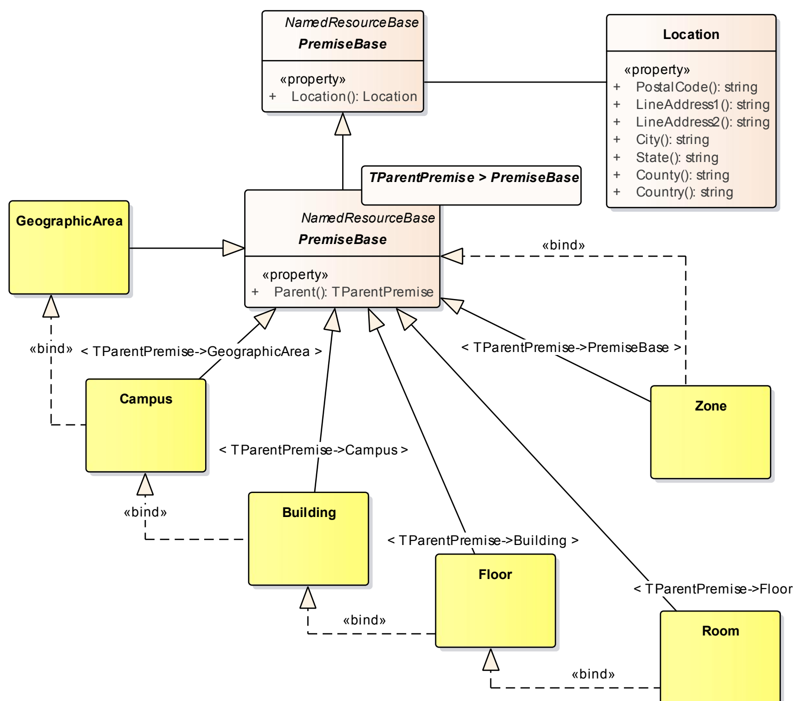

Figure 2. Premise Resource Models

Copyright (c) IARIA, 2016. ISBN: 978-1-61208-497-8

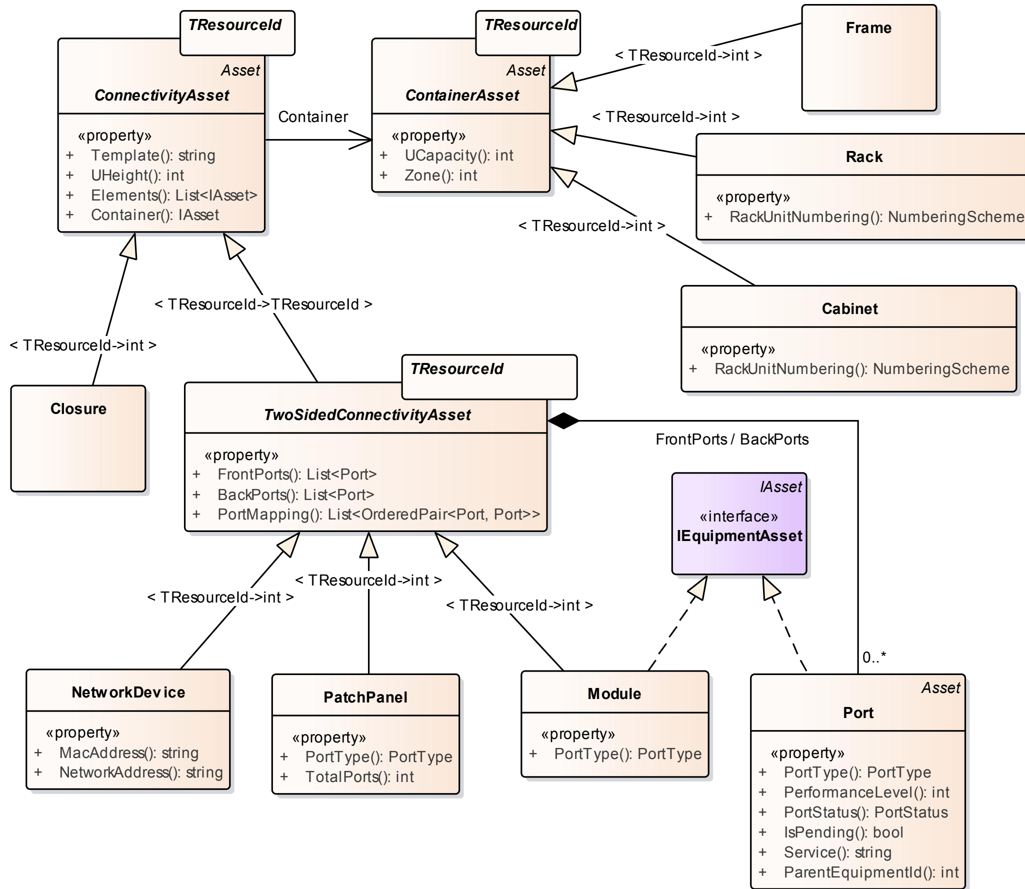

Figure 3. Telecommunication Assets Resource Models

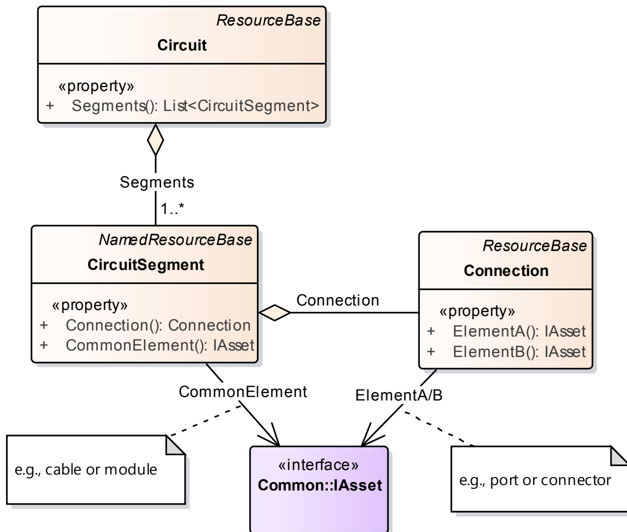

Figure 4. Connectivity Models


```
{
  "rackUnitNumbering":140002,
  "uCapacity":46,
  "zone":1,
  "position":2,
  "name":"Rack A123",
  "description":"Rack hosting instaPATCH panels",
  "concreteAssetType":"Rack (7 ft - 45U)",
  "concreteAssetTypeId":10,
  "parentId":26
}
```


Figure 5. A JSON Representation of a Rack Resource

software (events). These notification resource types are supported by the AIM Standards and are modeled as shown in Figure 6. The figure also includes activities that technicians must carry out, such as establishing connections between assets, activities that in turn trigger alarms and events, or are created as a reaction to system-generated events.

Copyright (c) IARIA, 2016. ISBN: 978-1-61208-497-8

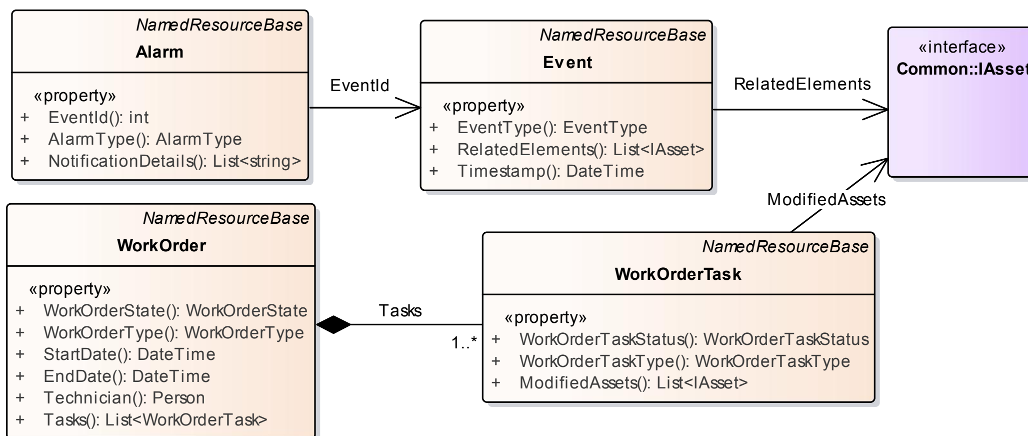

Figure 6. Notification and Activity Models

D. *Modeling Large Varieties of Hardware Devices*

The telecom asset model presented in Figure 3 depicted the categories that define all or most physical devices seen in network infrastructure. However, actual hardware components have specialized features that are vendor-specific or that describe some essential functionality that the components provide. Such specialized attributes must be incorporated in the model for supporting the Add (POST) and Update (PUT) functionality of the RESTful services that expose these objects to the integrators. The main challenge is how to support such a large variety of hardware devices without having to expose too many different service endpoints for each of these specialized types.

According to the Richardson Maturity Model for REST APIs [7], which breaks down the principal ingredients of a REST approach into three steps, Level 1 requires that the API be able to distinguish between different resources via URIs; i.e., for a given resource type there exists a distinct service endpoint to where HTTP requests are directed. For querying data using HTTP GET, we can easily envision a service endpoint for a given resource *category* – as per the models described above. For example, there will be one URI for modules, one for closures, one for patch panels, etc. However, when creating new assets, we have to be very clear about which concrete entity or device type we want to create, and for this, we must provide the device-specific data. Since these features are not inherent to all objects that belong to that category, specialized models must be created – e.g., as derived types from the category models that encapsulate all relevant device-specific features.

For example, one of CommScope connectivity products that falls under the category of Closures is the SYSTIMAX 360™ Ultra High Density Port Replication Fiber Shelf, 1U, with three InstaPATCH® 360 Ultra High Density Port Replication Modules [15] – a connectivity solution for high-density data centers that provides greater capacity in a smaller, more compact footprint. These closures come in a variety of configurations and aside from the common closure attributes (position, elements, capacity, etc.) other properties are relevant from a provisioning, connectivity, and circuit tracing perspective. Such properties include Orientation of the sub-modules, Location in Rack, Maximum Ports, and Port Type, as shown in the class diagram in Figure 7.

An alternative to using an inheritance model would be to create distinct types for each individual physical component that could be provisioned by the AIM system, but given the significant overlap of common features they can be consolidated and encapsulated in such a way that derived specialized models can be employed in order to increase code reusability, testability, and maintainability. The differentiation between the various hardware components that map to the same specialized type can be managed, for example, via metadata associated with that data type (e.g., the `AllowedObjectTypeAttribute` in Figure 7).

This approach saves us from having to define one data type per physical device type and furthermore, allows accessing a variety of devices that fall under the same category, using the same URI – as described in the next sub-section.

E. *Benefits of the Proposed Model*

The models proposed in this paper are closely following the categories and entities outlined by the ISO/IEC standards. However, given the structural models presented here and taking advantage of available technology-specific constructs and frameworks, select design features exist that confer certain advantages to these models, to their usage, and the integration capabilities for the services that expose them, with direct impact on performance, maintainability, testability, and extensibility.

Copyright (c) IARIA, 2016. ISBN: 978-1-61208-497-8

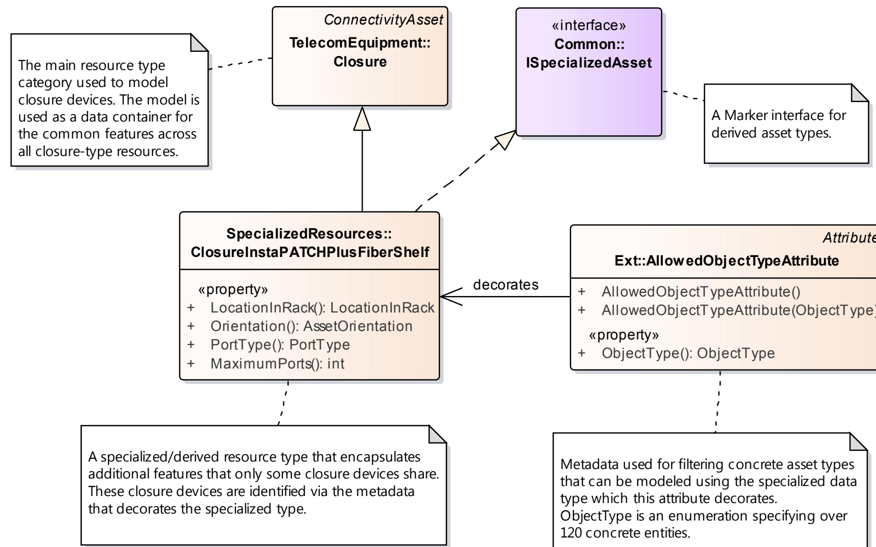

Figure 7. A Sample Specialized Closure with Additional Properties

- ✓ *<u>Simplified URI scheme</u> based on resource categories rather than specialized resource types*. This allows clients to access classes or categories of resources rather than having to be aware of - and invoke - a large number of URIs dictated by the large variety of hardware devices modeled. This also confers the API a high degree of stability and consistency even when the system is enhanced to provision new hardware devices.
- ✓ *<u>Reduced chattiness</u> between client application and services when <u>querying resources</u>* (`GET`). This benefit is directly related to the URI scheme mentioned above, since a single HTTP request can retrieve all resources of that type (applying the Liskov substitution principle [8]), even when multiple sub-types exist.
- ✓ *<u>Reduced chattiness</u> between client application and services when <u>creating complex entities</u>* (`POST`) *by supporting composite resources*. In some cases, the hardware device construction itself requires the API to support creating a resource along with its children in a single step (see Section IV.B for details). Child elements can be specified as part of the main resource to be created, or they can be omitted altogether, while – in the case of the imVision API - the Validation and Composition frameworks would take care of filling in the missing sub-resources based on predefined composition and default initialization rules.

  Table 2 captures just a few but noteworthy metrics regarding the request counts and sizes for creating a complete resource of a specialized `PatchPanel` type.
- ✓ *<u>Opportunity for automation</u> when creating and validating composite resources*. Aside from considerably reducing the size of the request body given the option to omit child elements when adding new entities - as is the case for the imVision API – by employing frameworks that support metadata-driven automation, the API will ensure that the generated resource object reflects a valid hardware entity – with all required sub-elements. For the API consumers, this reduces the burden of knowing all the fine details about how these entities are composed and constructed. In some cases, the number of child elements to be created in

TABLE II. POST REQUEST METRICS FOR QUATTRO PANEL (A PATCHPANEL RESOURCE)

Metric	Scenario	Value
Number of POST Requests	Without Support for Composite Resources	**31**: 1 for the Panel, 6 for the child Modules, and 6x4 for the ports
	With Support for Composite Resources	**1**: a single request for the Panel with its Modules (under `Elements`), with each Module being itself a composite resource containing 4 ports each, specified under the `FrontPorts` property of each Module
POST Request Body Size	With Explicit Children Included	21,449 bytes
	With No Children Specified (i.e. relying on the Framework to populate default elements)	572 bytes

the process depends on properties that the main resource may expose (e.g. `TotalPorts`) – which client applications will have to specify if the property is marked as [`Required`], but the composing port sub-elements may be omitted from the request body, as they will be automatically created and added.

- ✓ *Extensible model as new hardware devices are introduced*. New models can easily be added to the existing specialized resources or as a new subtype. The interface for querying the data (`GET`) will not change. The design for adding and updating resources follows the Open/Closed principle [8], so that new types, properties, and rules will be added or extended but existing ones will not change, ensuring contract stability.

III. A PROPOSED LAYERED ARCHITECTURE FOR AIM API INTEGRATION SERVICES

A. Adding Integration Capabilities to an AIM System

As per the Standards document guidelines, the AIM Systems should follow either an HTTP SOAP or a RESTful service design. Regardless of the service interface choice, there are several options for designing the overall AIM system. A common yet robust architectural style for software systems is the layered architecture [6] [11], which advocates a logical grouping of components into layers and ensuring that the communication between components is allowed only between adjacent or neighboring layers. Moreover, following SOLID design principles [8], this interaction takes place via interfaces, allowing for a loosely coupled system [9], easy to maintain, test, and extend. This will also enable the use of dependency injection technologies such as Unity, MEF, AutoFac, etc., to create a modular, testable, and coherent design [12].

CommScope's imVision system was built as a standalone web-based application, to be deployed at the customer's site, along with its own database and various middleware services that enable the communication between the hardware and the application. Relying on the current system's database, the RESTful Services were added as an integration point to the existing system. The layered design of this new service component is shown in Figure 8 with the core component – the resource model discussed earlier, shown as part of the domain layer. The system also utilizes - to a very limited extent - a few components from the existing imVision system that encapsulate reusable logic.

Several framework components were used, most notably the Validation component, which contains the domain rules that specify the logic for creating and composing the various entities exposed by the API. These rules constitute the core component upon which the POST functionality relies. Along with the resource composition and validation engine, they constitute in fact a highly specialized rule-based system that makes extensive use of several design and enterprise integration patterns that will be discussed next.

B. Patterns and Design Principles

The various patterns and principles [6] [8] [9] employed throughout the design and implementation of the imVision API system are summarized in Table 3. The automation capabilities baked into the imVision API mentioned earlier, that support creating composite resources, are a direct realization of the Content Enricher integration pattern used in conjunction with the Builder, Composite, and Specification software design patterns. From a messaging perspective, all requests are synchronous and only authorized users (Claim Check pattern) are allowed to access the API.

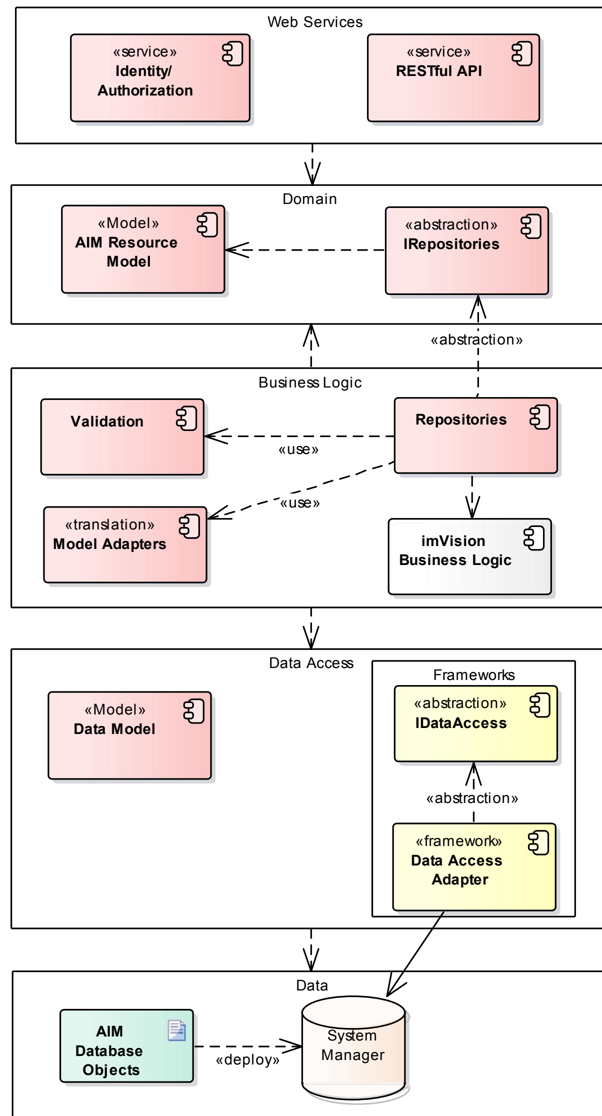

Figure 8. The Layered Architecture of the imVision AIM API

TABLE III. DESIGN PATTERNS AND PRINCIPLES EMPLOYED

Design Patterns		
Type	***Category***	***Pattern Name***
Design Patterns	Creational	Abstract Factory, Builder, Singleton, Lazy Initialization
	Structural	Front Controller, Composite, Adapter
	Behavioral	Template Method, Specification
Enterprise Application Patterns	Domain Logic	Domain Model, Service Layer
	Data Source Architectural	Data Mapper
	Object-Relational Behavioral	Unit of Work
	Object-Relational Metadata Mapping	Repository
	Web Presentation	Front Controller
	Distribution Patterns	Data Transfer Object (DTO)
	Base Patterns	Layer Supertype, Separated Interface
Enterprise Integration Patterns	Messaging Channels	Point-to-Point Channel Adapter
	Message Construction	Request-Reply
	Message Transformation	Content Enricher Content Filter Claim Check Canonical Data Model
	Composed Messaging	Synchronous (Web Services)
Design Principles		
SOLID Design Principles	**S**ingle Responsibility Principle (SRP) **O**pen/Closed **I**nterface Segregation **L**iskov Substitution (in conjunction with co- and contra-variance of generic types in .NET) **D**ependency Inversion (Data Access and Repositories are injected using MEF and Unity)	

IV. A FEW CHALLENGES AND SOLUTIONS

A. *Handling POST Requests for Large Numbers of Specialized Resource Types with Few URIs*

Simplified URI schemes have the benefit of providing a clean interface to consumers, without having to introduce a myriad of URIs, one per actual hardware device supported by the AIM system.

As shown in Section II, the different representation of these resources are grouped by category, while specific details are handled using *custom* JSON *deserialization* behavior injected in the HTTP transport pipeline [2] [13]. Since all resources must specify the concrete entity type they represent (under the `ConcreteAssetTypeId` property), the custom deserialization framework can easily create instances of the *specialized* resource types based on this property, and pass them to the appropriate controller (one per URI/resource category).

The impact on performance is negligible given the use of a lookup dictionary of asset type ID to resource type, which is created only once (per app pool lifecycle) based on metadata defined on the model. Even if new specialized resource types are added, the lookup table will automatically be updated at the time the application pool is instantiated (restarted), ensuring the inherent extensibility of the custom deserialization framework.

This way, whether a user would like to create a "360 iPatch Ultra High Density Fiber Shelf (2U)" or a "360 iPatch Modular Evolve Angled (24-Port)" [15], even though these two hardware devices map to two different specialized types in the imVision API resource model, they are both resources of type `PatchPanel`. Therefore, a POST request to create either of these will be sent to the same URI: `http://[host:port/app/]PatchPanels`

This means that the same service components (controller and repository) will be able to handle either request but the API would also be aware of the distinction between these two different object instances, as created by the custom deserialization component.

B. *Adding Support for Composite Resources*

Hardware components are built as composite devices, containing child elements, which in turn contain sub-child entities. For example, the Quattro Panel contains six Copper Modules with each module containing exactly four Quattro Panel Ports. To realize these hardware-driven requirements and avoiding multiple POST requests, while preserving the integrity of the device representation, a rule-based composition representation model was used in conjunction with the Builder design pattern applied recursively.

The composition rules for the Quattro Panel and its module sub-elements are shown in Figure 9 (The strings represent optional name prefixes for the child elements.).

```
//...
{ ObjectType.QuattroPanel24Port, new CompositionDetail<ModuleCopperModule, int, ModuleValidator>(ObjectType.CopperModule, "Module", 6) },
//...
{ ObjectType.CopperModule, new CompositionDetail<PortBasicPort, int, PortValidator>(ObjectType.QuattroPanelPort, "Port", 4) },
//...
```

Figure 9. Composition Rules for Quattro Panel and Its Child Elements of Type Copper Module

C. A Functional, Rule-Based Approach for Default Initializations and Validations of Resources

Given the large number of specialized resources to be supported by CommScope's imVision API and the even larger number of business rules regarding the initialization and validation of these entities, a functional approach was adopted. This rendered the validation engine into a rule-based system: there are *composition rules* (above), default *initialization rules*, and *validation rules* (below) – which refer to both simple as well as complex properties that define a resource. Following the same example of Quattro Panel used earlier, an important requirement for creating such resources is the labeling of ports and their positions, which must be continuous across all six modules that the panel contains.

Figure 10 shows a snapshot of the rules defined for this type of asset: Figure 10 (a) shows the initialization rules whereas Figure 10 (b) shows some of the validation rules. In both cases, the programming constructs like the ones shown make heavy use of lambda expressions as supported by the functional capabilities built into the C#.NET programming language [14], demonstrating the functional implementation approach adopted for the imVision API.

Among some of the reasons worth mentioning for taking the functional route are a more robust, concise, reusable, and testable code, and minimizing side effects from object state management and concurrency. Explicit goal specification, central to the functional programming paradigm, confers clarity and brevity to the rule definitions, both evident in the code samples provided hereby.

```
result[ObjectType.QuattroPanel24Port] = new Lazy<Dictionary<string, IPropertyInitDetail>>(() =>
        new Dictionary<string, IPropertyInitDetail>
        {
            [nameof(PatchPanel.UHeight)] = new PropertyInitDetail<int>(() => 1),
            [nameof(PatchPanel.TotalPorts)] = new PropertyInitDetail<int>(() => 24),
            [nameof(PatchPanel.PortType)] = new PropertyInitDetail<PortType>(() => PortType.Rj45),
            [nameof(PatchPanel.Elements)] =
              new PropertyInitDetail<IEnumerable<IAsset>, PatchPanel>((r) =>
              DefaultElementsFactory.GenerateElements(ObjectType.QuattroPanel24Port,
              6, r, new List<Action<Module, PatchPanel>>
              {
                (module, panel) => module.FrontPorts.ForEach(x =>
                {
                  x.Name = ((module.Position - 1)*4 + x.Position).ToString("00");
                  x.Position = ((module.Position - 1)*4 + x.Position);
                }),
              })),
        });
```

Figure 10. (a) Default Initialization Rules Sample

```
result[ObjectType.QuattroPanel24Port] = new List<IValidationRule>
{
  new ValidationRule<PatchPanelPreTermCopperPanel>(nameof(PatchPanelPreTermCopperPanel.TotalPorts),
                x => x.TotalPorts == 24, "Total ports must be equal to 24."),
  new ValidationRule<PatchPanelPreTermCopperPanel>(nameof(PatchPanelPreTermCopperPanel.PortType),
                x => x.PortType == PortType.Rj45),
  new ValidationRule<PatchPanelPreTermCopperPanel>(nameof(PatchPanelPreTermCopperPanel.Elements),
                x => x.Elements != null && x.Elements.Count() > 1 && x.Elements.Count() <= 6,
                "Invalid ELements Count: There must be at least one but no more than 6 modules."),
  new ValidationRule<PatchPanel>(nameof(PatchPanel.Elements),
                x => x.Elements.OfType<ModuleCopperModule>().Select(
                      y => y.Position).Distinct().Count() == x.Elements.Count,
                "Invalid Module Positions"), //distinct positions of modules validation
  new ValidationRule<PatchPanel>(nameof(PatchPanel.Elements),
                x => x.Elements.OfType<ModuleCopperModule>().SelectMany(
                      y => y.FrontPorts.Select(z => z.Name)).Distinct().Count() == x.Elements.Count * 4,
                "Non-distinct port names."),
   //more rules ...
};
```

Figure 10. (b) Validation Rules Sample

Copyright (c) IARIA, 2016. ISBN: 978-1-61208-497-8

V. CONCLUSION

Modeling large varieties of telecommunication assets can be a challenging task, even more so if other applications intend to integrate with one or more systems that automate the management of such complex telecommunication enterprise infrastructure. The benefits entailed by the standardization of modeling entities managed by such systems are significant, as they facilitate a common understanding of the AIM system in general and the elements it exposes, their functional features, and their internal makeup. ISO/IEC proposed such standardization for a more systematic and unified modeling of AIM systems. This paper took further steps to present detailed models and the relationships between them using design artifacts modeled via UML (Unified Modeling Language). Using inheritance, composition/aggregation, and generic typing, a hierarchical resource model was designed and shown to be extensible and fit for representing telecom assets, connectivity, premises, organizational elements, and system notifications – as they relate to any AIM-centric domain.

Although the focus of the 18598/DIS draft ISO/IEC Standards document is to unify the representation of network connectivity assets, the motivation behind this specification is to facilitate custom integration solutions with AIM systems. Given the challenging nature of integration in general, building AIM systems with integration in mind is essential. Extensibility, scalability, rigorous and stable interface and model design, and performance through adequate technology adoption are important goals to consider. For this reason, the present paper also introduced the layered architecture adopted by CommScope's imVision API, targeting the management of telecommunications infrastructure.

Emphasis was placed on the Standards-recommended RESTful architectural style, while technology specifics were succinctly described to show how they helped align the system's design and functionality with the AIM standards requirements. Various design and implementation aspects were elaborated along with a selection of key benefits, such as dynamic resource composition, custom serialization to support consistent handling of similar resources, efficient POST request construction and network traffic, and a simple URI scheme despite large varieties of specialized resources.

Finally, a very brief overview of a rule-based engine for resource initialization and validation was described, along with some implementation details that highlight aspects of the functional programming paradigm employed by key components of CommScope's imVision API.

VI. REFERENCES

[1] Automated Infrastructure Management(AIM) Systems– Requirements, Data Exchange and Applications, 18598/DIS draft @ ISO/IEC.

[2] G. Block, et. al., “Designing Evolvable Web APIs with ASP.NET”, ISBN-13: 978-1449337711.

[3] R. Daigneau, “Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services”, Addison-Wesley, 1st Edition, 2011, ISBN-13: 078-5342544206.

[4] T. Erl, “Service-Oriented Architecture (SOA): Concepts, Technology, and Design,” Prentice Hall, 2005, ISBN-13: 978-0131858589.

[5] E. Evans, “Domain-Driven Design: Tackling Complexity in the Heart of Software,”, 1st Edition, Prentice Hall, 2003, ISBN-13: 978-0321125217.

[6] M. Fowler, “Patterns of Enterprise Application Architecture,” Addison-Wesley Professional, 2002.

[7] M. Fowler, “The Richardson Maturity Model”. [Online]. Available from http://martinfowler.com/articles/richardsonMaturityModel.html [retrieved: March 2016].

[8] G. M. Hall, “Adaptive Code via C#: Agile coding with design patterns and SOLID principles (Developer Reference),”, Microsoft Press, 1st Edition, 2014, ISBN-13: 978-0735683204.

[9] G. Hohpe, B. Woolf, “Enterprise Integration Patterns; Designing, Building, and Deploying Messaging Solutions,” Addison-Wesley, 2012, ISBN-13: 978-0321200686.

[10] J. Kurtz, B. Wortman, “ASP.NET Web API 2: Building a REST Service from Start to Finish,” 2nd Edition., 2014, ISBN-13: 978-1484201107.

[11] Microsoft, “Microsoft Application Architecture Guide (Patterns and Practices),” Second Edition, Microsoft. ISBN-13: 978-0735627109. [Online] Available from: https://msdn.microsoft.com/en-us/library/ff650706.aspx [retrieved: March 2016].

[12] M. Seemann, “Dependency Injection in .NET,” Manning Publications, 1st Edition., 2011, ISBN-13: 978-1935182504.

[13] J. Webber, “REST in Practice: Hypermedia and Systems Architecture,” 1st Edition, 2010, ISBN-13: 978-0596805821.

[14] T. Petricek, J. Skeet, “Real-World Functional Programming: With Examples in F# and C#”, Manning Publications; 1st edition, 2010, ISBN-13: 978-1933988924.

[15] CommScope Enterprise Product Catalog. [Online] Available from: http://www.commscope.com/Product-Catalog/Enterprise/ [retrieved March 2016].

A Research Roadmap for Test Design in Automated Integration Testing of Vehicular Systems

Daniel Flemström
SICS Swedish ICT AB
Västerås, Sweden
danielf@sics.se

Thomas Gustafsson
Scania CV AB
Södertälje, Sweden
thomas.gustafsson@scania.com

Avenir Kobetski
SICS Swedish ICT AB
Kista, Sweden
avenir@sics.se

Daniel Sundmark
Mälardalen University
Västerås, Sweden
daniel.sundmark@mdh.se

***Abstract*—An increasing share of the innovations emerging in the vehicular industry are implemented in software. Consequently, vehicular electrical systems are becoming more and more complex with an increasing number of functions, computational nodes and complex sensors, e.g., cameras and radars. The introduction of autonomous functional components, such as advanced driver assistance systems, highlight the foreseeable complexity of different parts of the system interacting with each other and with the human driver. It is of utmost importance that the testing effort can scale with this increasing complexity. In this paper, we review the challenges that we are facing in integration testing of complex embedded vehicular systems. Further, based on these challenges we outline a set of research directions for semi-automated or automated test design and execution in integration testing of vehicular systems. While the discussion is exemplified with our hands-on experience of the automotive industry, much of the concepts can be generalised to a broader setting of complex embedded systems.**

Index Terms*—*Software Testing*; *Automotive Systems*; *Embedded Systems*; *Integration Testing

I. Introduction

Electrical systems in modern vehicles grow increasingly complex and software intensive. With additional concerns, like increasing requirements on autonomy and safety, integration- and system-level testing becomes more and more challenging. At full-vehicle integration level, in addition to the testing performed in actual vehicles, different types of electrical system lab testing is undertaken. Typically, in lab testing, use-case based functional (and to some extent non-functional) test cases are executed by means of hardware-in-the-loop (HIL) or software-in-the-loop (SIL) based integration testing platforms.

This current practice of test design and execution comes with a number of drawbacks. Test execution is costly and time-consuming, particularly if done manually. However, when test cases are scripted and automatically executed, testing tends to be repetitive and static. Moreover, the fact that test cases typically focus on one functional part at a time (without considering the potential interactions or interference between functions) may not account for realistic operating conditions.

In this paper, we address the question of whether it is possible to use automated test design, execution and analysis to test complex systems in general, and automotive and vehicular systems in particular, more effectively without exhausting the test resources. First, based on our experience with integration-level testing at a number of different vehicular Original Equipment Manufacturers (OEMs), we list a number of challenges that need to be addressed. Next, partially based on recent results in software and system testing, we outline a research roadmap for integration- and system-level testing of vehicular systems. In particular, we identify and describe five research directions for test design, automated test sequence generation and verdict analysis, that directly address the listed challenges.

II. Background

Over the last decades, embedded systems have been subjected to a rapid increase in complex functionality, and there are no indications suggesting that this trend will change in a foreseeable future. This is especially true for automotive systems, where emission regulations and *advanced driver assistance systems (ADAS)* energize the development. The functionality of ADAS depend on an increasing amount of sensor data. This leads to an increasing number of situations where human operators will no longer be in control. Based on input from various sensory sources, different functional components will interact and sometimes compete with each other and the operator. This adds flavour to the already non-trivial challenge of testing a product in its entirety.

The remainder of this section discusses state of the practice based on the collective experience from complex system integration testing in general, and integration-level testing of a number of vehicular OEMs (Volvo Construction Equipment (VCE), Bombardier Transportation (BT) and Scania) in particular. Although there are differences in particular details, the principles and challenges remain common.

A. State of the Practice

Today, testing is the primary means of assessing the quality of embedded systems. Testing is typically done at several stages throughout system development, ranging from unit-level testing of functions in isolation, to integration testing of fully interconnected systems. Ordered by the rising level of integration, testing is normally conducted in model-in-the-loop (MIL), SIL, HIL test environments, and full scale product tests. Typically, this is done through use-case based test sequences, partly derived from system requirements and partly reflecting

Copyright (c) IARIA, 2016. ISBN: 978-1-61208-497-8

the test engineers' domain knowledge. This type of testing is commonly referred to as scenario-based testing [1].

In scenario-based testing, test cases are designed using a divide-and-conquer-based approach following the breakdown of system requirements into smaller functional entities (which is in accordance with recent textbook guidelines for functional test design [2], [3]). Test cases are typically constructed by means of sequences of (a) input or stimuli to the *system under test (SUT)*, (b) delays allowing the SUT to reach some desired state, and (c) assertions that compare expected behavior with the actual SUT behavior. Based on the outcome of its assertions, a test case, when exercised, can render three different verdicts: a *passed* test case is a test case with no violated assertions, a *failed* test case is a test case where the expected behavior does not match the behavior of the system under test (ideally indicating a fault in the target system), and an *aborted* test case is a test case where some test case action cannot be performed (ideally indicating a fault in the test case or the test environment). Once having been designed, implemented, and incorporated in the test suite, test cases are typically repetitively executed without much variation.

While manual testing is invaluable for quality assurance, especially when new functionality is introduced, it is expensive and not suitable for regression testing (i.e., following up on how software progresses over time whenever a new part of the system has been added to the SUT or an existing system has changed due to a bug fix or requirement update). For the latter purpose, test cases are—to an increasing extent—scripted to allow for automated batched execution. Analyzing which of the existing test cases should be executed in each regression test session is often a manual and time consuming process.

III. Current Practice: Limitations and Challenges

Although scripted scenario-based testing does ensure that requirements are exercised and covered during testing, it comes with a number of limitations. Below, we identify a set of challenges not addressed satisfactorily by current practice. It should be noted here that some manual testing practices (e.g., exploratory testing [4]) do address some of these challenges. However, the focus of this paper is on systematic automated techniques, primarily due to the fact that such techniques are more likely to scale with increasing complexity.

A. Ch1 - Lack of functional interference

The procedure of decomposing requirements into smaller functional units, and using these units in isolation as the basis for test case design is fundamental in managing an otherwise overwhelming complexity. However, this procedure for requirement decomposition, and subsequent test design also prevents the assessment of undesirable interference between functional entities. Such assessment is particularly important as the functions grow increasingly complex and dependent on several interacting subsystems. Small disturbances in one system component may propagate into fault conditions in another, not to mention the case when different functional entities are in conflict or competing over the same resource.

B. Ch2 - Inefficient resource usage

In order to make sure that each test has the right preconditions for making a correct verdict, test scripts are generally executed one by one, while the SUT's state is reset between the test scripts. In addition to the time overhead that is needed to set up and shut down each test case, delays are often encoded explicitly in test scripts to represent correct timing behavior (e.g., response times) of the SUT. Sequential test execution means that delay times can never run in parallel. Not only does this poorly represent real operating conditions, where several independent functions (e.g., cruise control, radio, and turn indicator lights) can be active at the same time, but it also leads to inefficient use of testing resources.

C. Ch3 - Inflexible and tedious test case encoding

Integration level test cases are often coded manually into a scripting language. One reason for this is the wish to keep a tight control over the order and timing of stimuli to the SUT to avoid incorrect test verdicts, i.e., verdicts (typically pass or abort) caused by incorrect test case implementation rather than the actual behavior of the SUT. In practice, this means that a test engineer needs to consider in detail not only what is the expected behavior for a certain function, but also how to put the SUT in a state where such behavior is supposed to be manifested. This mixes up two different views of the system, namely the design (and subsequent implementation) of test stimuli sequences and test oracles, making the task of a test engineer even more challenging. As a consequence, the development of integration test cases is a rather complex and time-consuming process, which severely limits the number of test cases that can be implemented and maintained.

D. Ch4 - Unnecessarily limited coverage

Any given functional requirement could in theory be tested by a large (if not infinite) set of concrete test cases. However, the current practice of hard-coding of scenarios into scripted test cases limits the potentials of variability in testing to only the hard-coded cases. Once designed and scripted, a scenario-based test case is typically kept and repetitively executed without much or any variation. While this has the advantage of detecting differences between software versions (e.g., for regression testing), it limits the testing to a very small portion of the vast set of imaginable real-life situations that the system could be subjected to, while the rest is left entirely unexplored. For example, the functionality of a hazard light could potentially benefit from being tested not only when cruising at low speed, but also in situations like: i) when the vehicle is idling, ii) while driving on a highway in winter conditions, iii) while applying brakes, etc. In practice though, most situations fall outside of the chosen set of test scenarios being considered important enough to encode and maintain.

E. Ch5 - Inadequate requirements

Previous work suggests that test cases are often based on positively stated requirements defining how the system should

Copyright (c) IARIA, 2016. ISBN: 978-1-61208-497-8

behave during normal operation [5]. While this provides valuable confirmation with respect to the system's fitness for use in the normal case, there are results indicating that focusing on normal requirement-based cases might not be the best strategy when trying to maximize fault-detection [6], [7]. In fact, there exist many examples of accidents caused not by a software bug in the classical sense, but rather by incorrect or missing requirements, caused by a failure to consider important operational states or environmental conditions [8].

Also, a common problem in testing in general is that although some non-functional robustness testing is done, a disproportionally large part of testing is concerned mainly with functional requirements [9]. Related, there is a gap between requirements engineering (RE) and testing activities. In some cases, requirement documents are followed, but more often such documents must be complemented with the domain knowledge and understanding of what needs to be tested that the test engineer possesses. This is quite common in the industrial practice, also noted in, e.g., [9]. Another issue, which also depends on the above mentioned gap between RE and testing, is the practice of developing test cases for the same functionality independently, on different integration levels. As the development rate of new functionality is steadily increasing, such approach to system level testing seems insufficient.

F. Ch6 - The "smart product" challenge

In general, embedded products become increasingly intelligent, while testing methods are lagging behind. In the vehicular world, this is best highlighted by ADAS functionality, which introduces additional challenges to the software development process and to testing in particular. Advanced algorithms are to an increasing extent replacing human decision making. While human drivers might and do make mistakes, there is close to zero-tolerance when it comes to autonomous functions doing so. Thus, ADAS-type functions must be tested even more extensively. Also, their interactions with other functionality in the vehicle, as well as with the driver, must be taken into consideration. Most ADAS functionality is triggered by the surrounding environment and the different situations that a vehicle may be subjected to. This is a challenge in a laboratory setting (e.g., HIL), where the environment must be represented in some way.

IV. RESEARCH DIRECTIONS

Considering the above stated challenges, we believe that much can be gained from moving beyond the prevailing script-based integration test design and execution practice, which is primarily focused on one single function or even a part of a function (use case) in isolation. In general, we envision a situation where automatically executable test sequences are generated or derived in a semi-automated or fully automated fashion. These test sequences are derived based on high-level test design rationales that address the above challenges. The sequences can be automatically executed and their verdict can be automatically analysed and reported. Below, we list a number of research directions that serve to push the integration testing of vehicular systems towards this vision.

First, considering the inflexible and tedious test case encoding challenge (Ch3), we suggest to draw a clear line between generation of test stimuli sequences (i.e., timed and ordered sequences of input data to be fed to the SUT), and encoding of expected test responses (i.e., how the SUT should respond given a certain sequence of events or stimuli). This research direction is elaborated in the subsection on **Separation of Concerns**. Second, concerning *test sequence generation (TSG)*, we identify three distinct rationales that directly address challenges Ch1, Ch2, Ch4, and Ch6. These rationales are **Functional Interference**, focusing on the extent to which the interaction between features is covered during testing, **Environmental Coverage**, focusing on the extent to which aspects of the intended environment of the vehicle are covered during testing, and **Diversity**, focusing on the extent to which test cases and test suites are different from one another. Each of the above listed research directions are discussed in detail below. Third, modeling of expected responses has as its goal to robustly assess whether the response of the SUT to a variety of (automatically generated) test sequences is adequate (i.e., should yield a *pass* verdict) or not (i.e., should yield a *fail* verdict). In software testing, this problem is known as **the Oracle Problem** [10], and we address it in a separate subsection below. Referring to Ch5, note that any requirements that should be testable need to be reflected in test oracle models in an adequate way.

A. Separation of Concerns

As mentioned in challenge Ch3, the traditional way of encoding stimuli and verdicts into one executable unit makes the resulting test cases both less flexible and more difficult to analyze. Consequently, we believe that it is important to clearly separate: a) the question of how to combine test stimuli into effective and feasible test sequences, and b) the question of which test assertions to make in order to produce a test verdict, and when to evaluate these assertions.

With such separation in place, test stimuli modeling can be seen as a special case of *model-based testing (MBT)*, where the modeling scope, according to Utting et al.'s taxonomy [11], is limited to input-only. In other words, when reasoning about TSG, there is no explicit need to consider the test output, i.e., the actual verdict analysis. Instead, one can focus on questions such as what kind of input should be considered for guiding SUT through its possible states, and how this input should be modeled and selected into the actual test sequences to achieve effective and efficient testing.

Considering the other side of the problem, i.e., how to reach a test verdict given an automatically generated test sequence, with this approach test engineers can focus more explicitly on exactly which preconditions are needed to trigger a test oracle function, and which results it should produce under different conditions. While appropriate models need to include both input and output aspects, the input part does not actually drive the state progression. Rather, it describes the state(s) in

Copyright (c) IARIA, 2016.

which the SUT needs to be in order to perform meaningful test evaluations. Ideally, such models should clearly reflect the requirements on functionality.

While separation of concerns contains relatively few research questions in itself, it is central not only for reducing test case generation complexity, but also for research on each of the separated parts. An important research direction here is to find appropriate interfaces between the separated parts, which in turn influences the choice of modeling formalisms in the two distinct cases. Some preliminary results have already been pointed to in this direction [12], but much work remains.

B. Functional interference

In real operating conditions, functional interference is the norm rather than an exception [13]. Different parts of a complex product's functionality are typically being engaged simultaneously and independently from each other. Taking an automotive application as an example, a lane shift may involve both turn indicator and acceleration functions. At the same time, the climate control system may be trying to keep temperature at some set point. Including ADAS into consideration, e.g. blind spot assist or adaptive cruise control functionality may also be active during the lane shift.

One goal of integration testing is to test how different functions work together. However, use-case based testing typically focuses on one function at a time, and the concurrent activation of other functions is only a side-effect of reaching a testable system state. Clearly, if interactions between functions are not *systematically* tested, there is a risk of missing important errors. In fact, function or subsystem interactions are responsible for a growing portion of accidents in complex systems [8]. Also, it is important to carefully consider possible interactions between autonomous functionality and human actions. In fact, in fields where automation was adopted early, such as aviation, such interactions are known to contain a certain risk [14].

Consequently, it is desirable to allow for testing of several functions run in parallel. In the ideal case that any combination of functions that in some way affect one another or a common functional or non-functional resource was tested together, Ch1 could be removed from the above list of challenges. Conversely, the usage of testing resources (Ch2) would be much more efficient if all fully independent function combinations could be batched together into a single run. However, there are a number of questions to solve before reaching that situation. For example, what is an adequate runtime environment for parallel testing of potentially dependent functions? Possibly, ideas could be drawn from the field of parallel computing. However, they should be adapted to the specifics of integration testing of embedded systems, e.g. complex functional dependencies, hard real-time constraints, wide range of different users and operational contexts, safety criticality, etc., see also [13].

Since testing resources are normally limited, the right mix between interacting and independent functionality will likely be an important design trade-off that should be considered by TSG algorithms. Also, infeasible combinations of functions, i.e., those leading to test abortion, must somehow be avoided.

Different types of functionality need to be modeled in a suitable way, such as actuator-triggered (driver controls), sensor-triggered (autonomous responses), and failure-triggered.

C. Environmental coverage

An important factor that affects the operation of embedded systems, often neglected in scenario-based testing, is the impact of the surrounding environment on a system's performance and operation. This aspect is important not only by extending the notion of test coverage with a new dimension (thus addressing Ch4), but it increases in significance with the growing "smartness" of embedded products (Ch6). The more autonomous functionality a system contains, the more important it is to anticipate and test possible situations that the system can be subjected to. This need has recently been formulated by Alexander et al. [15], as a situation coverage metric for autonomous systems.

The idea is to describe real-life situations by partitioning them into a number of constituent components, or environmental aspects, each of which consists of a number of discrete (and typically mutually exclusive) elements, or possible values. Returning to the automotive world for an example, such components could be the topography of the road (uphill, downhill, or negligible inclination), road surface conditions (snow, ice, rain, dry), surrounding traffic (pedestrians, queues, highway, platooning, etc.), intersection types (3-road, right-turn, left-turn, straight driving, etc.), driver condition (alert, tired, using phone, etc.), and so on.

Once the environmental components have been identified, they can be combined into more or less complex situations and tested together with several active functional elements. For example, a vehicle can be driving uphill on a snowy highway, conducted by a sleepy driver that makes a lane shift and presses down the acceleration pedal, while blind spot assist and adaptive cruise control functionalities are activated.

Evidently, situation coverage poses similar research questions as the ones discussed in the previous subsection, perhaps the most obvious being the choice of appropriate environmental models, with respect to, e.g., abstraction level, modeling formalism, TSG tools, etc. Further, since the number of possible combinations of situational components and functions seems to suffer from combinatorial explosion, it is not realistic to believe that every possible aspect will be tested in each test run. Thus, combinatorial strategies on selecting relevant situations with respect to the testing objectives are needed [16].

D. Diversity

A recent research direction in software testing investigates the effects on fault detection and coverage of the extent to which test cases are different from each other. In particular, *diversity* metrics based on information theory have been used for this purpose [17]. Diversity is typically defined as a metric between 0 and 1 indicating the distance between two test cases, or sets of test cases. Several studies indicate that increased test case diversity has a positive effect on the ability to discover faults. For instance, Mondal et al. [18] used diversity as a

complement to traditional test adequacy criteria and found that combining diversity with other criteria yields better fault detection rates. Feldt et al. [17] define a diversity metric for sets of test cases, which allows for search-based selection methods to work on entire sets of test cases. There are however several different definitions of diversity, and thus, ways of measuring it. Examples of such metrics are based on the normalised compression distance (NCD) between the textual representation of a variability model of the test cases [17], [19], the euclidian distance between input/output vectors [20] or features thereof [21], and weighted combinations of metrics for different properties of the measured items [22].

Extending the concepts of e.g test input/output diversity, we may address the first challenge, Ch1, by using diversity as a criterion to ensure that a generated set of test cases involves as different functions as possible. Further, using diversity measures to guide the selection of test cases between regression testing sessions would also address Ch4 to avoid that the same, or too similar, tests are being executed from time to time, thus yielding a better coverage over time.

Given the promising results in the mentioned studies (although largely focused on unit testing), diversity stands out as an attractive optimization criteria for a test stimuli generator as discussed in Section IV-A. There are however numerous challenges with this approach, including the level of detail of the available information and to what extent such information can be efficiently retrieved. Further challenges include how to apply diversity in an event-based testing environment (e.g., considering timing and parallelism), and how to best combine diversity with other metrics (e.g. requirements, environmental coverage, or functional interference). Research applicable definitions of diversity that are effective on integration level with respect to the different challenges listed in Section III is therefore needed. Finally, the computational effort when calculating the diversity measures needs to be addressed in order to be suitable for a test sequence generator.

E. Test oracle modeling

A *test oracle* is a mechanism that, given a certain input and the SUT's response to that input, can state whether this actual response is in accordance with the expected response (i.e., if the test passes or not). In software testing, the construction of such a mechanism is known as the oracle problem. The oracle problem is relatively unexplored and inherently difficult to address [10]. In practice, since complete oracles would require an exhaustive and correct representation of the expected behaviour of the SUT, only partial oracles are possible to construct, typically encoded as assertions in test cases.

As mentioned above, we believe that test oracles should be modeled as passive analysis mechanisms, expressing how a SUT should behave given a certain sequence of stimuli. Condition models [23] and guarded assertions [24] are two examples of initial attempts to address this problem. However, further development is needed to reach practical applicability.

Ideally, oracle models should be formal enough to allow for automatic translation into test code. Further, they should have clear and human-readable links to the functional requirements they represent. This would promote requirement traceability and ability to reason about logical relations between tests and requirements, reducing the gap between these disciplines (Ch5). Also, automation of the logic behind oracles should then be possible, supporting oracle analysis either online, e.g., by testing in a HIL environment, or offline, e.g., by a verification algorithm. Balancing between informal and formal requirement models is thus an intriguing research topic.

Also, human-readable models should be developed early on, allowing their reuse through different stages of the testing process. Modeling patterns on different abstraction levels, and unambiguous conversion between the different models and the test code will likely be needed for any such approach to be applicable. Related, structured English grammars, together with patterns to facilitate writing requirements, suitable for automated checking of system and requirements conformance, have been proposed in several papers [25], [26], [27].

Finally, addressing the remaining parts of Ch5, oracle models reflecting non-positive or non-functional requirements will be needed. The challenge is further complicated by the immaturity of the RE field in this respect. There is a clear need for research both on basic RE in this direction, and next on enriching the results of such basic research into the more applied question of oracle modeling.

F. Summary

Above, five research directions are outlined for test design in integration testing of vehicular systems. Considering all these research directions combined, one could envision the following integration test design, execution and analysis process: **First**, requirements are (manually) encoded as abstract and passive test oracles using human readable and intuitive patterns. **Second**, based on these oracles, but also considering other test design rationales like functional interference, diversity and/or environmental coverage, test stimuli sequences are automatically generated. **Third**, the test sequences are executed on the system under test, and the oracles automatically analyze the system response in order to produce an aggregated test verdict.

Naturally, the outlined process is only one possible way of moving forward and should be revised as testing research and practice progress. Further, it is important to relate the industrial experience, to recent academic advances, drawing inspiration from the broader test research field, e.g. [28], [29].

V. Conclusions

In this paper, a number of challenges facing the discipline of integration testing is outlined, based on the authors' experiences from industrial vehicular systems. The increasing autonomy and complexity of modern vehicles, which leads to a high level of functionality interaction, and in consequence complex emergent behavior, need to be accounted for in integration testing. This poses an additional burden on the already restrained testing resources. In addition, it becomes more difficult to reason about system behavior, which makes coverage an even more important aspect to consider.

Copyright (c) IARIA, 2016. ISBN: 978-1-61208-497-8

The challenges are considered in a discussion on interesting research directions. Firstly, as in other areas of software engineering, concerns should be separated where possible. It is our claim that generation of test stimuli sequences can and should be separated from the actual oracle function. This will allow a focus on the different parts separately, generating appropriate models for each. On the oracle side, it is important to capture requirements at the right level of abstraction, ideally transforming them into more formal models that can be verified by a test run. When it comes to test stimuli, there is a need for appropriate models that capture actual situations to which a vehicle can be subjected, with several interacting functions being active simultaneously.

An important topic for research is how to increase the variability of what is tested. One answer to this question is to make the test input space as diverse as possible. However, vehicles are complex creatures operating in complex environments, and so the modelling of possible inputs becomes challenging, not to mention the question of what is meant by diversity and how to select appropriate test stimuli in order to produce a diverse test suite. Taking functional interaction and coverage of environmental conditions into consideration may provide some answers to the above questions.

ACKNOWLEDGMENT

This work was supported by The Swedish Innovation Agency (Vinnova) through grant 2015-04816, and the Swedish Knowledge Foundation through grant 20130258.

REFERENCES

[1] A. Bertolino, E. Marchetti, and H. Muccini, “Introducing a reasonably complete and coherent approach for model-based testing,” *Electr. Notes Theor. Comput. Sci.*, vol. 116, pp. 85–97, 2005.

[2] M. Young and M. Pezze, *Software Testing and Analysis: Process, Principles and Techniques*. John Wiley & Sons, 2005.

[3] P. Ammann and J. Offutt, *Introduction to Software Testing*, 1st ed. New York, NY, USA: Cambridge University Press, 2008.

[4] J. Itkonen, M. V. Mäntylä, and C. Lassenius, “The role of the tester’s knowledge in exploratory software testing,” *IEEE Transactions on Software Engineering*, vol. 39, no. 5, pp. 707–724, May 2013.

[5] L. M. Leventhal, B. Teasley, D. S. Rohlman, and K. Instone, “Positive test bias in software testing among professionals: A review,” in *Selected papers from the Third International Conference on Human-Computer Interaction*, ser. EWHCI ’93. London, UK, UK: Springer-Verlag, 1993, pp. 210–218. [Online]. Available: http://dl.acm.org/citation.cfm?id=646181.682601

[6] L. M. Leventhal, B. E. Teasley, and D. S. Rohlman, “Analyses of factors related to positive test bias in software testing,” *Int. J. Hum.-Comput. Stud.*, vol. 41, no. 5, pp. 717–749, Nov. 1994. [Online]. Available: http://dx.doi.org/10.1006/ijhc.1994.1079

[7] A. Causevic, R. Shukla, S. Punnekkat, and D. Sundmark, “Effects of negative testing on tdd: An industrial experiment,” in *International Conference on Agile Software Development, XP2013*, H.Baumeister and B. Weber, Eds. Springer, June 2013, Date accessed: 2016-06-09. [Online]. Available: http://www.es.mdh.se/publications/2771-

[8] N. G. Leveson, “System safety in computer-controlled automotive systems,” *SAE transactions*, vol. 109, no. 7, pp. 287–294, 2000.

[9] Z. A. Barmi, A. H. Ebrahimi, and R. Feldt, “Alignment of requirements specification and testing: A systematic mapping study,” in *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*. IEEE, 2011, pp. 476–485.

[10] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, “The oracle problem in software testing: A survey,” *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 507–525, May 2015.

[11] M. Utting, A. Pretschner, and B. Legeard, “A taxonomy of model-based testing approaches,” *Software Testing, Verification and Reliability*, vol. 22, no. 5, pp. 297–312, 2012.

[12] T. Gustafsson, M. Skoglund, A. Kobetski, and D. Sundmark, “Automotive system testing by independent guarded assertions,” in *Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on*, April 2015, pp. 1–7.

[13] M. Broy, “Challenges in automotive software engineering,” in *Proceedings of the 28th international conference on Software engineering*. ACM, 2006, pp. 33–42.

[14] T. B. Sheridan and R. Parasuraman, “Human-automation interaction,” *Reviews of human factors and ergonomics*, vol. 1, no. 1, pp. 89–129, 2005.

[15] R. Alexander, H. Hawkins, and A. Rae, *Situation coverage – a coverage criterion for testing autonomous robots*. Department of Computer Science, University of York, 2 2015, vol. Report number YCS-2015-496.

[16] C. Nie and H. Leung, “A survey of combinatorial testing,” *ACM Comput. Surv.*, vol. 43, no. 2, pp. 11:1–11:29, Feb. 2011. [Online]. Available: http://doi.acm.org/10.1145/1883612.1883618

[17] R. Feldt, S. Poulding, D. Clark, and S. Yoo, “Test set diameter: Quantifying the diversity of sets of test cases,” *arXiv preprint arXiv:1506.03482*, 2015.

[18] D. Mondal, H. Hemmati, and S. Durocher, “Exploring test suite diversification and code coverage in multi-objective test case selection,” in *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on*. IEEE, 2015, pp. 1–10.

[19] R. Feldt, R. Torkar, T. Gorschek, and W. Afzal, “Searching for cognitively diverse tests: Towards universal test diversity metrics,” in *Software Testing Verification and Validation Workshop, 2008. ICSTW’08. IEEE International Conference on*. IEEE, 2008, pp. 178–186.

[20] P. Bueno, W. E. Wong, and M. Jino, “Improving random test sets using the diversity oriented test data generation,” in *Proceedings of the 2nd international workshop on Random testing: co-located with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*. ACM, 2007, pp. 10–17.

[21] R. Matinnejad, S. Nejati, L. C. Briand, and T. Bruckmann, “Simcotest: a test suite generation tool for simulink/stateflow controllers,” in *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016 - Companion Volume*, 2016, pp. 585–588, Date accessed: 2016-06-09. [Online]. Available: http://doi.acm.org/10.1145/2889160.2889162

[22] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer, “Object distance and its application to adaptive random testing of object-oriented programs,” in *Proceedings of the 1st international workshop on Random testing*. ACM, 2006, pp. 55–63.

[23] A. Ray, I. Morschhaeuser, C. Ackermann, R. Cleaveland, C. Shelton, and C. Martin, “Validating automotive control software using instrumentation-based verification,” in *Automated Software Engineering, 2009. ASE’09. 24th IEEE/ACM International Conference on*. IEEE, 2009, pp. 15–25.

[24] G. Rodriguez-Navas, A. Kobetski, D. Sundmark, and T. Gustafsson, “Offline analysis of independent guarded assertions in automotive integration testing,” in *High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS), 2015 IEEE 12th International Conferen on Embedded Software and Systems (ICESS), 2015 IEEE 17th International Conference on*, Aug 2015, pp. 1066–1073.

[25] A. Mavin, P. Wilkinson, A. Harwood, and M. Novak, “Easy approach to requirements syntax (ears),” in *Requirements Engineering Conference, 2009. RE’09. 17th IEEE International*. IEEE, 2009, pp. 317–322.

[26] M. Autili, L. Grunske, M. Lumpe, P. Pelliccione, and A. Tang, “Aligning qualitative, real-time, and probabilistic property specification patterns using a structured english grammar,” *Software Engineering, IEEE Transactions on*, vol. 41, no. 7, pp. 620–638, July 2015.

[27] P. Filipovikj, M. Nyberg, and G. Rodriguez-Navas, “Reassessing the pattern-based approach for formalizing requirements in the automotive domain,” in *Requirements Engineering Conference (RE), 2014 IEEE 22nd International*. IEEE, 2014, pp. 444–450.

[28] M. J. Harrold, “Testing: a roadmap,” in *Proceedings of the conference on the future of software engineering*. ACM, 2000, pp. 61–72.

[29] A. Orso and G. Rothermel, “Software testing: a research travelogue (2000–2014),” in *Proceedings of the on Future of Software Engineering*. ACM, 2014, pp. 117–132.

Case Study: Becoming a Medical Device Software Supplier

Kitija Trektere, Fergal McCaffery
Regulated Software Research Centre & Lero,
Dundalk Institute of Technology
Dundalk, Co.Louth, Ireland
e-mail: {kitija.trektere, fergal.mccaffery}@dkit.ie

Garret Coady, Matteo Gubellini
BlueBridge Technologies
3015 Lake Drive, Citywest, Dublin 24, Ireland
e-mail: {garretcoady, matteogubellini}@bluebridgetech.com

***Abstract*—Today many software development companies are restructuring their business model to enter the medical device domain. The reason for this change is that significant opportunities exist within the healthcare industry and particularly in relation to the usage of software within this domain. However, in order to become either a medical device software supplier or manufacturer there are challenges to overcome. This paper describes a case study of an Irish software development company that in 2014 decided to change their business model to enable them to become a medical device software supplier. The paper provides an account of their journey from being an plan-driven automotive software supplier to securing software development contracts from leading medical device manufacturers. This involved them having to re-design and re-structure their software development approach to meet both the demands of medical device standards and medical device multinational third party software selection criteria.**

Keywords-MDevSPICE® framework; software development process; medical device software; medical device software development; agile practices; agile software development practices.

I. INTRODUCTION

In 2015, the medical device (MD) global market was "valued at $228 billion, up from $164 in 2010 and projected to reach $440 billion by 2018" "at approximately 4.4% compound annual growth rate per year" [1]. The leaders in the MD market are the US having 38% of the global value of this market followed by China with a market valued at $48 billion with western Europe having almost 25% of the global market [1]. However, to become a MD supplier for the industry takes significant time and resources as there are many obstacles that need to be overcome.

This paper presents a case study of an Irish software development company *BlueBridge Technologies* (BBT). Their journey started in 2014 when BBT decided to embark upon becoming a MD software supplier and at that moment they had no regulatory requirements in place, in fact a key question they asked at that stage was *"what are the standards we need to implement and in what order?"*. This paper presents how with the help of academic MD researchers' regulations were put in place through undergoing an MDevSPICE® assessment and outlining the challenges that might arise in the near future.

The rest of this paper is organized as follows. Section II, describes the background of BBT and the current situation in the MD industry. Section III, outlines the challenges BBT faced in order to become a MD software supplier. Section IV, describes the approaches followed to become a MD software supplier. Section V, outlines given that BBT have satisfied the regulations they wish to further refine and improve their software development processes to make them more efficient. Section VI, describes first steps taken in order to improve their current lifecycle process. Finally, Section VII provides a conclusion and future work.

II. BACKGROUND OF THE COMPANY

BTT was founded in 2006 – initially formed upon the closure of the Irish based development operations of Magna Automotive, and today employs 19 people with 8 of them working as software developers. BBT are currently working on 7 different projects with 5 of them involving developing the software component for another organization's product. Their current customers include pharmaceutical and multinational MD companies.

The main reason why software development companies wish to enter the MD domain is because of the expansion of the MD industry in the past few years therefore providing many opportunities for others to enter into this industry. The MD industry is largely research and development driven.

Software increasingly performs an essential role in the provision of healthcare services [2]. This is particularly reflected in the importance that software now plays in medical diagnoses and treatment [3]. The level of software functionality in MDs and the complexity of that software has substantially increased [4]. The MD regulatory environment has been extended to include more focus on software. For example, the latest amendment to the Medical Device Directive [5] recognizes that standalone software can be classified as a MD in its own right. Consequently, a significantly increased proportion of software applications will now be classified as MDs and must be developed in a regulatory compliant manner [6]. Medical records are increasingly being stored in electronic form. The use of Electronic Medical Record (EMR) systems in the USA by physicians increased from 18.2% in 2001 to 48.3% in 2010

Copyright (c) IARIA, 2016. ISBN: 978-1-61208-497-8

[7]. The adoption of EMR systems could produce efficiency and safety savings of $81 billion annually and improve prevention of medical diseases [8]. Use of Mobile devices in health care is increasing. “By 2017, mobile technology will be a key enabler of healthcare delivery reaching every corner of the globe” [9].

III. CHALLENGES BBT NEEDED TO OVERCOME TO BECOME A MD SOFTWARE SUPPLIER

To become a MD software supplier there were regulations and standards that needed to be adhered to. This required processes to be defined in accordance with these standards and regulations and then for objective evidence to be obtained demonstrating the implementation of the defined processes. For BBT, the starting point was to gain an understanding of three main standards. The paragraph below briefly outlines the standards that BBT familiarized themselves with before starting to define their MD software development processes.

A. *ISO 13485:2006*

“This International Standard specifies requirements for a quality management system that can be used by an organization for the design and development, production, installation and servicing of medical devices, and the design, development, and provision of related services” [11].

ISO 13485 is in practice necessary for any MD company. It details the requirements for the Quality Management System (QMS) for MDs.

B. *IEC 62304*

“This standard defines the life cycle requirements for Medical Device Software. The set of processes, activities, and tasks described in this standard establishes a common framework for medical device software life cycle processes” [10].

IEC 62304 covers the development process for medical device software. This standard is harmonised with the requirements of ISO 13485 and therefore complements it by adding the specifics required for MD software.

However, IEC 62304 interfaces with ISO 13485 in two areas: software inputs and system integration. The software inputs are generated from the system (or subsystem) level requirements, while IEC 62304 explicitly does not cover system level activities, in particular design validation.

C. *ISO 14971:2009*

“This International Standard was developed specifically for medical device/system manufacturers using established principles of risk management. For other manufacturers, e.g., in other healthcare industries, this International Standard could be used as informative guidance in developing and maintaining a risk management system and process”[12]. “This International Standard deals with processes for managing risks, primarily to the patient, but also to the operator, other persons, other equipment and the environment” [12].

The area of regulatory standards and the recording of documentation associated with their implementation was new to BBT. Therefore, BBT engaged with both standards consultants and an academic research group (the RSRC, our research centre) specializing in MD software development research. This assisted BBT to fast-track the initial steps to becoming a MD software supplier.

IV. APPROACH TO BECOME A MD SOFTWARE COMPANY

When BBT reached out to the RSRC, we knew that this was an ideal company to become involved with in regards to performing research into how software companies could make the transition to becoming MD software suppliers.

A. *Embark on MDevSpice® assessment*

First of all, it was essential to understand BBT’s current position in regards to their software development processes. We decided to perform an MDevSPICE® [13] assessment. MDevSPICE® is a framework assessment model where all MD software standards and processes are brought together into one place with software engineering best practices. MDevSPICE® was developed in the RSRC. Then, this framework assessment model was utilized in BBT to assess the current situation.

Below we describe what happened next in regards to both the assessment and BBT’s subsequent journey to becoming a MD software supplier. As BBT were used to developing in a plan-driven manner and this would enable them to develop medical device software in compliance with the V-model we did not investigate agile practices at this stage.

A) Assessment conducted: Given that MDevSPICE® consists of 23 processes we selected the most appropriate 10 processes from the MDevSPICE® model to assess BBT against (see Table I).

It was agreed upon discussion with BBT that only the most foundational processes would be assessed. Therefore, the following 10 out of the 23 MDevSPICE® processes were chosen to be assessed over two onsite days at BBT.
The 10 processes chosen for the assessment were selected because they provided coverage of IEC 62304 and also provided an important system level input into the software development process. The system level process was important as BBT’s software formed part of an overall medical device system comprising hardware and software and electronics. The processes were agreed upon during a meeting with senior management at BBT and the assessment team prior to the assessment. The order of the processes assessed was important as it is important to follow the medical device software development lifecycle. Therefore, systems requirements were a very natural place to start.

Copyright (c) IARIA, 2016. ISBN: 978-1-61208-497-8

TABLE I. PROCESSES OF MDEVSPICE®

MD System Lifecycle Processes	*MD Software Lifecycle Processes*	*MD Support Processes*
Project Planning	Software Development Planning	Configuration Management
Project Assessment and Control	Software Requirements Analysis	Software Release
Risk Management	Software Architectural Design	Software Problem Resolution
Stakeholder Requirements Definition	Software Detailed Design	Software Change Request Management
System Requirement Analysis	Software Unit Implementation. and Verification	Software Maintenance
System Architectural Design	Software Integration and Integration Testing	
System Integration	Software System Testing	
System Qualification Testing	Software Risk Management	
Software Installation		
Software Acceptance Support		

Below is outlined the process assessment schedule: we assessed 5 processes on each day (See Table II).

TABLE II. DAY 1 AND DAY 2 OF ASSESSMENT PROCESS

Onsite Assessment Day 1
System Requirements Analysis
Software Development Planning
Software Requirements Analysis
Software Architectural Design
Software Detailed Design
Onsite Assessment Day 2
Software Unit Implementation & Verification
Software Integration & Integration Testing
Software System Testing
Software Risk Management
Software Configuration Management

Each process was assessed by two MDevSPICE® assessors in an interview with at least two members of BBT being present in each interview. Prior to the interviews both the schedule and the names of the BBT staff members that would be involved in each process interview was agreed. It was very important to ensure that access was provided to the most relevant staff for each interview session as otherwise the assessment would not have been as accurate as possible.

Each of the 10 interviews lasted approximately one hour and involved one assessor asking BBT staff a set of scripted questions related to that process area. The second assessor used a tool to record detailed responses from the interviewees with both assessors using the tool to enable each question to be scored as "Fully Achieved", "Partially Achieved" or "Not Achieved". In addition to the usage of predefined scripted questions, additional questions were also asked that were specific to BBT.

B) Findings produced: The MDevSPICE® assessors at the end of Day two returned back to the RSRC and went through each process together, discusssing the observations and notes from the assessment. As a result of performing the assessment we provided BBT with a set of strengths, issues and recommendations to address those issues across each of the assessed processes.

The MDevSPICE® assessment provided coverage over a number of different MD software related standards. Figure 1 shows a breakdown of the coverage provided for each of the different standards from assessing 10 of the 23 MDevSPICE® processes. One of the key objectives of BBT Management was to gain an understanding in relation to the state of their current development processes against IEC 62304, as this is the main MD software process standard, processes were selected from MDevSPICE® that featured heavily in IEC 62304. The exception to this was System Requirements Analysis but this was deemed to be a critical process to examine as BBT would be performing software development for an overall MD system. Therefore, it is essential that they have an efficient process in place for System Requirements Analysis as otherwise everything that occurs afterwards within the development lifecycle will be impacted.

From looking at Figure 1 it can be seen that the 10 processes assessed provided: 59% coverage of IEC 62304; 2% of ISO 80002-1 [14] (this technical report relates to how ISO 14971 may be applied within software); 16% of the FDA's Guidance for off the shelf software [15]; 1% of the FDA's Guidance for premarket submissions [16]; 20% of the FDA's Guidance for validation of software [17]; 1% of ISO 13485 and 1% of software engineering best practice standards.

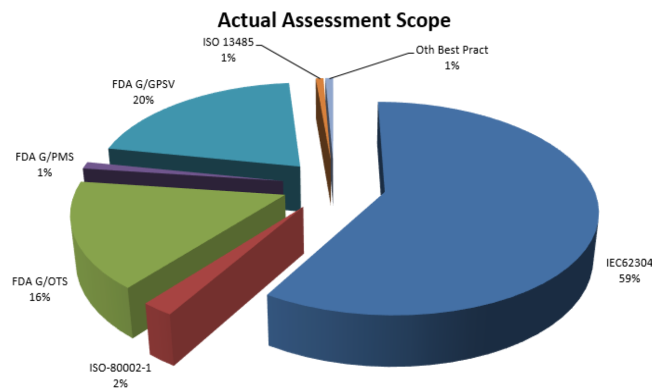

Figure 1. Scope of the BBT Assessment.

C) Implementing the recommendations: In order to assist BBT to implement the recommendations in a timely manner BBT took the following steps:

a) Brought in consultants to assist with the implementation of QMS 13485.

b) Recruited an engineer from a leading MD manufacturer who possessed considerable experience in developing MD software development and in particular MD risk management expertise to put in place a risk management strategy in line with ISO 14971.

c) Engaged with a notified body organisation to prepare them for an official audit in IEC 62304 and subsequently perform the audit. This enabled a successful IEC 62304 audit to be achieved in a timely manner.

D) Actions taken by BBT:

a) Gained Certification in IEC 62304.

b) The IEC 62304 audit was performed against one project using a plan driven approach.

c) BBT implemented the required MD standards for medical device software suppliers.

The main criteria MD manufacturers use for selecting a MD software supplier is that organizations should have IEC 62304 in place. At this stage BBT now have not only satisfied this criteria but surpassed it in that they not only adopted IEC 62304 but were certified against it and also have adopted IEC 13485, ISO 14971 and the FDA Guidance documentation for: Off the shelf software, Premarket Submissions and Validation of MD software. Therefore, at this stage BBT were ready to obtain contracts as a MD software supplier company.

V. WHAT HAPPENED NEXT?

Once BBT became a MD software supplier they noticed the significant attention within the MD field. MD software manufacturers started to get in contact and invite tenders for various projects. In fact, to date they have worked on a number of MD software development projects for different types and sizes of manufacturers. Therefore, the overhead required to implement the necessary standards was starting to pay dividends. However, now that the opportunities clearly are out there it is noticeable that BBT now want to move to the next phase of their MD software development journey and not only develop software in line with the MD standards but their ambition is now to increase the efficiency of their MD software development. Therefore, they wish to improve their software development processes even further and implement more regulatory standards in relation to security etc. The key driver to take a step further is that BBT now are undertaking challenging projects and are developing MD software for multinational MD companies they have much more to achieve in their journey. BBT have agreed to work with researchers from the RSRC to introduce MD software development best practices that will increase the efficiency of their MD software development.

A. *Challenges for such large projects*

However, as with every new project there are associated challenges and this is increased when embarking upon a *fixed price* project, therefore if the project is delayed or runs into some other difficulties, BTT is liable in relation to the budget. Another challenge is the *tight timeframe* where *strict milestones* have to be achieved in addition to the achievement of appropriate documentation to satisfy *regulatory deliverables*. Additionally, BBT would also like to excel in being able to *facilitate change* during the lifecycle of the project as this is something that is challenging in traditional MD software development. A very positive aspect of BBT's current approach is that they engage in regular interaction with their customers. Therefore, receiving feedback and making sure that the right MD software is developed from the very start of the development.

B. *What is the current status of BBT development process lifecycle?*

Currently BBT is developing software in a plan driven way through using the V-model [18]. When following a V-model the testing is planned in parallel with the corresponding development phase and the planning for verification and validation of the product is emphasized from the very beginning. To date, V-model has been proven to be the best fit when developing MD software in compliance with the regulations [19]. However, in order to improve the efficiency of their software development new software practices should be explored that have proven successful in the development of safety-critical software in association with researchers from the RSRC.

Before introducing a new lifecycle it is crucial to perform an assessment in order to establish how the current software development process should be improved/changed.

VI. ASSESSMENT PROCESS AND RESULTS

The following subsections will describe the high-level assessment process completed in BBT in 2016.

A) Assessment Process

The Software development process assessment was performed at BBT before deciding what new practices would be most suitable for BBT. We met up with the CEO of the company, project manager/developer (who had has experience of agile software development), and a developer who specialized in Android software development. The meeting was also attended by the R&D manager/Systems Risk engineer and the QMS manager. The assessment was based on previously scripted open-ended questions that related to many different areas of the company as well as the software development process.

B) Results

a) Currently BBT have several standards in place, such as IEC 62304, ISO 13485, ISO 9001 and ISO 14971. In their software development process they make use of

various tools in areas such as project management, testing and integration. One of their main drivers for adopting new best practice software development methods is to streamline even further their already succcessful practices for interacting with customers. BBT view this as being key to delivering safe regulatory compliant software that fully meets the customer requirements and works within the intended environment, thereby decreasing the chances of expensive rework, particularly on fixed price projects.

b) Additionally, they wish to develop metrics such as problem tracking, code coverage, defects found, defects closed etc.

c) In the past, BBT was open to changes and customers were able to introduce them whenever they wanted without consequences to the overall budget, time. However, today the process has become more structured. BBT now ensures that a formal change request document is in place specifying what happens if a change occurs within a previously signed project.

d) BBT at the moment is not making use of any principle software design techniques however, they plan to introduce architecture diagrams and design patterns.

e) BBT previously have developed software in a plan driven manner and lately they have decided to integrate some agile practices into their development process..

f) Currently almost 80% of testing is automated and 20% is done manually. If the percentage of manual testing could be decreased further – the overall development process could become faster.

g) One of the team members mentioned that due to the new lifecycle approach where agile practices are introduced, there could be a challenges regarding integrating the QMS with the development process and achieving the necessary regulatory documentations.

h) At present their current process incorporates only two agile practices, they are: short iterations (every two weeks) and continuous integration.

i) BBT is also planning to provide their team with the training needed in order to work in an environment where MD software is developed in an agile way. The team will be provided with training in regards to MD software, agile practices and mobile app development.

j) Some team members will be provided with support to change towards adopting a more agile software development process.

C) Recommendations

Our advice to BBT is to integrate more agile practices into their current MD software development so that the software is developed efficiently in regular iterations and can be presented to the customer on a regular basis and facilitate change. Based upon a mini-literature review performed, the following agile practices have been cited as being used to develop software successfully for safety critical/medical domains:

a) Acceptance test-driven development (ATDD) [20].

b) Automated Tests/Automated unit testing [21].

c) Code Reviews / Peer Reviews [22].

d) Coding Standards [20][23].

e) Continuous integration (CI) [20][23][24].

f) Open Workspace [20][25].

g) Scrum [26].

h) Test-driven development (TDD) [20][27].

VII. CONCLUSION AND FUTURE WORK

This paper described a case study of a journey taken by an Irish software development company, moving from developing automotive software to developing MD software. We described how through adopting and implementing MD standards they now have become a MD software supplier. The key contribution of this research work was to enable an organization such as BBT to overcome the challenging and resource intensive learning process of understanding what standards should be put in place in order to become a medical device software supplier. Secondly, it was important to be able to provide guidance as to the order in which practices should be put in place so that previously implemented practices will not need to be overwritten later. Since becoming a MD software supplier many new opportunities have become available. However, BBT now wish to further improve their software development processes in order to become more efficient and to be able to satisfy new challenges that could rise from undertaking new multinational MD manufacturer's projects. The authors of the paper have provided a list of agile practices that have been cited to be well suitable for safety critical/medical domain.

In the future, we plan to investigate agile practices that are applicable for the MD software industry in greater detail by performing an extensive literature review and industry survey. Further, we will work with BBT to integrate the most applicable agile practices into their current software development lifecycle.

ACKNOWLEDGMENT

This research is supported by the Science Foundation Ireland Research Centres Programme, through Lero - the Irish Software Research Centre (http://www.lero.ie) grant 10/CE/I1855 & 13/RC/20194.

REFERENCES

[1] J. Cunningham, B. Dolan, D. Kelly, and C. Young, "Medical Device Sectoral Overview," Galway City and County Economic and Industrial Baseline Study, 2015.

[2] C. Abraham, E. Nishihara, and M. Akiyama, "Transforming healthcare with information technology in Japan: A review of

policy, people, and progress," Int. J. Med. Inform., vol. 80, no. 3, pp. 157–170, 2011.

[3] S. Hanna, R. Rolf, A. Molina-Markham, P. Poosankam, K. Fu, and D. Song, "Take two software updates and see me in the morning: The Case for Software Security Evaluations of Medical Devices," in The 2nd USENIX conference on Health security and privacy, pp.6, 2011.

[4] S. R. Rakitin, "Coping with Defective Software in Medical Devices," pp. 40–45, 2006.

[5] EC, "Directive 2007/47/EC of the European Parliament and of the Council concerning medical devices," Official Journal of the European Union. Official Journal of the European Union, Brussels, Belgium, p. 35, 2007.

[6] F. McCaffery, J. Burton, A. Dorling, and V. Casey, "Software Process Improvement in the Medical Device Industry," in Software Engineering Encyclopaedia, P. Laplante, Ed. New York: Francis Taylor Group, 2010, pp. 528 – 540.

[7] C.-J. Hsiao, E. Hing, T. C. Socey, and B. Cai, "NCHS Health E-Stat Electronic Medical Record/Electronic Health Record Systems of Office-based Physicians: United States, 2009 and Preliminary 2010 State Estimates," Natl. Cent. Heal. Stat., vol. 2009, no. December, p. 6, 2010.

[8] G. Hillestad, R., Bigelow, J., Bower, A. and F. Girosi, "Can Electronic Medical Record Systems Transform Health Care? Potential Health Benefits, Savings, And Costs," Health Aff., vol. 24, no. January, pp. 1103–17, 2016.

[9] "Mobile to Play a Significant Role in Healthcare as GSMA Research Predicts mHealth Market to be Worth US$23 billion by 2017," 2012. [Online]. Available: http://www.gsma.com/newsroom/press-release/mobile-to-play-a-significant-role-in-healthcare-as-gsma-research-predicts-mhealth-market-to-be-worth-us23-billion-by-2017/. [Accessed: 26-Mar-2016].

[10] BSI, "Medical device software- Software life-cycle processes, 62304:2006," Bs En 62304:2006, vol. 3. 2006.

[11] ISO, "ISO 13485: Medical Devices - Quality Management Systems - Requirements for Regulatory Purposes." Geneva, Switzerland, pp. 57, 2003.

[12] ISO, "ISO 14971 - Medical Devices - Application of Risk Management to Medical Devices." Geneva, Switzerland, pp. 82, 2009.

[13] F. McCaffery, M. Lepmets, and P. Clarke, "Medical Device Software as a Subsystem of an Overall Medical Device," in Proceedings of The First International Conference on Fundamentals and Advances in Software Systems Integration Medical, 2015, pp. 17–22.

[14] IEC, "IEC TR 80002-1 - Medical Device Software - Part 1: Guidance on the Application of ISO 14971 to Medical Device Software." Geneva, Switzerland, pp. 58, 2009.

[15] FDA, "Guidance for Industry - FDA Reviewers and Compliance on Off-The-Shelf Software Use in Medical Devices." USA, p. 26, 1999.

[16] FDA, "Guidance for the Content of Premarket Submissions for Software Contained in Medical Devices." USA, pp. 23, 2005.

[17] FDA, "General Principles of Software Validation ; Final Guidance for Industry and FDA Staff." USA, pp. 47, 2002.

[18] K. Forsberg and H. Mooz, "The Relationship of System Engineering to the Project Cycle," 12th INTERNET World Congr. Proj. Manag., pp. 12, June 1994.

[19] F. McCaffery, D. McFall, P. Donnelly, F. G. Wilkie, and R. Sterritt, "A Software Process Improvement Lifecycle Framework for the Medical Device Industry," in IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'05) Proceedings, pp. 8, 2005,

[20] S. Datta, P. Sarkar, S. Das, S. Sreshtha, P. Lade, and S. Majumder, "How many eyeballs does a bug need? An empirical validation of linus' law," in Lecture Notes in Business Information Processing, vol. 179 LNBIP, pp. 242–250, 2014,

[21] J. Grenning, "Launching extreme programming at a process-intensive company," IEEE Softw., vol. 18, no. 6, pp. 27–33, 2001.

[22] M. Bernhart, A. Mauczka, and T. Grechenig, "Adopting code reviews for agile software development," in Proceedings - 2010 Agile Conference, vol. 179 LNBIP, pp. 242–250, AGILE 2010.

[23] K. Beck, Extreme programming explained: embrace change. addison-wesley professional, 2000.

[24] P. Dahlem, Marc and Diebold, "Agile Practices in Practice - A Mapping Study Agile Practices in Practice - A Mapping Study -," in 18th International Conference on Evaluation and Assessment in Software Engineering, pp.30, MAY 2014.

[25] K. Beck, "Embracing change with extreme programming," Computer (Long. Beach. Calif)., vol. 32, no. 10, pp. 70–77, 1999.

[26] P. Abrahamsson, O. Salo, J. Ronkainen, and J. Warsta, Agile software development methods Review and analysis. ESPOO: VTT Publications 478, 2002.

[27] J. Rasmusson, The Agile Samurai, How Agile Masers Deliver Great Software. Texas, 2010.

A Specific Method for Software Reliability of Digital Controller in NPP

Young Jun Lee, Jong Yong Keum, Jang Soo Lee
I&C/HF Division
Korea Atomic Energy Research Institute
Daejeon, Korea
e-mail: yjlee426@kaeri.re.kr, jykeum@kaeri.re.kr, jslee@kaeri.re.kr

Young Kuk Kim
Department of Computer Science & Engineering
Chungnam National University
Daejeon, Korea
e-mail: ykim@cnu.ac.kr

***Abstract*— Most new controllers used in the safety systems of nuclear power plants have been developed using digital systems, and conventional analog controllers are also increasingly being replaced with digital controllers. Therefore, the importance of software that operates within the digital controller of a nuclear power plant is further increased. This paper describes a reliability evaluation method for the software to be used for a specific operation of a digital nuclear power controller. It is possible to calculate the software reliability when obtaining the failure rate and utilizing the existing calculation method. We attempt to achieve differentiation by creating a new definition of the fault, imitating the software fault using the hardware, and giving the consideration and weights for injection faults.**

Keywords- software reliability; digital controller in NPP; software life cycle; fault injection.

I. INTRODUCTION

To ensure the safety of software used in a Nuclear Power Plant (NPP), the Nuclear Regulatory Commission (NRC), the nuclear regulatory agency of the United States, has published its Software Review Plan (SRP) [1] and has required safety software to be developed according to the IEEE Standard 7-4.3.2 [2]. To meet these regulatory requirements, the software used in the nuclear safety field has been ensured through the development, validation, safety analysis, and quality assurance activities throughout the entire process life cycle from the planning phase to the installation phase [3]. However, this evaluation through the development and validation process needs a lot of time and money. In addition, a variety of activities, such as the quality assurance activities are also required to improve the quality of a software. However, there are limitations to ensure that the quality is improved enough. Therefore, the effort to calculate the reliability of the software continues for a quantitative evaluation instead of a qualitative evaluation.

In this paper, we propose a reliability evaluation method for the software to be used for a specific operation of the digital controller in an NPP. After injecting random faults in the internal space of a developed controller and calculating the ability to detect the injected faults using diagnostic software, we can evaluate the software reliability of a digital controller in an NPP. In Section 2, we introduce the reliability evaluation research for a nuclear software. A specific method for software reliability evaluation of a digital controller in an NPP is explained in Section 3. In Section 4, an experiment plan is suggested. Finally, we conclude the paper in Section 5.

II. RELIABILITY EVALUATION RESEARCH FOR A NUCLEAR SOFTWARE

Active research to assess the reliability of software in the field of a nuclear power plant has only recently progressed. It has been claimed that a quantitative calculation regarding the reliability of the software is impossible owing to the assumption that the software failure rate does not increase over time unlike in electronic components. Thus, focus on the amount of testing needs to be made to ensure the reliability of the predetermined target level rather than directly calculating the software reliability. Research methods that have been tailored for this purpose thus far include the Software Reliability Growth Model (SRGM) [4] and Bayesian Belief Net [5]. However, it is premature for these research methods to be applied directly to a site because of the specificity of an NPP. The applicability of such a method may be considered only after the result is stabilized and objectively proven. Software reliability assessment methods that have been researched regarding the current status of a nuclear power plant are as follows.

A. *Software Reliability Growth Model*

The SRGM is used to establish an assumption regarding whether the software reliability will be improved when a software failure by such a defect does not occur again by removing the defects that are inherent in the event of a software failure. There are two criteria: the Root Mean Square Error (RMSE) [4] and Average Error (AE) [4].

The RMSE is a measure commonly used when dealing with the difference between one of the model predictions based on observations in the real world. It is suitable to represent the precision. Each of the difference values is also referred to as a residual, and the mean square deviation is used to synthesize the residual as a measure. These criteria may be used to measure the difference between the actual value and the predicted value. Two formulas can be expressed as follows (1)(2):

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(c(k) - \hat{c}(k)\right)^2} \quad (1)$$

$$AE = \frac{1}{n}\sum_{i=1}^{n}\left|\frac{c(k)-\hat{c}(k)}{c(k)}\right| \times 100 \quad (2)$$

where n is the group number of the failure data, c(k) is the number of actual failures in each group of failure data, and ĉ(k) is the number of predicted failures. The smaller the RMSE and AE models, the more their predictive power increases.

Copyright (c) IARIA, 2016. ISBN: 978-1-61208-497-8

B. *Bayesian Belief Net*

Bayesian Belief Net (BBN) [15] is a methodology that leads to quantitative results by calculating and applying the laws of probability including Bayes probability. It models the relevant variables in the target system through a causal relationship, expresses the dependency degree of variables as a conditional probability, and inputs several observed evidences into the BBN model generated. The BBN consists of nodes indicated as circles on the graph, arcs between nodes, and a node probability table (or conditional probability table) of each node. Nodes represent variables included in the model, the arcs indicate a causal relationship between nodes. Each node has a number of states as random variables (for example, a state of "Yes" or "No"). The sum of the probability of the state value is 1. The node probability table associated with each node determines the connection strength between nodes, and is expressed as a conditional probability for each state of the parent node.

III. SPECIFIC METHOD FOR SOFTWARE RELIABILITY EVALUATION OF A DIGITAL CONTROLLER IN AN NPP

The software reliability methods we have seen thus far have been studied to apply to the software in an NPP, but there is actually no applied practice. SRGM can demonstrate that the reliability grows when failure data and the resolution case of a compete software exist. However, the adaptation data are not sufficiently secured and the BBN methodology has occupied much of the qualitative determination elements, and thus BBN methodology has a limitation in that it calculates the quantitative data. In order to overcome these disadvantages, we propose a specific method to obtain a quantitative value of the reliability of a software used in an NPP. Considerations in the proposed method are as follows.

First, the reliability evaluation formula uses the general reliability calculation method commonly used. This is the reliability calculation method for the electronic component. Applying this method to software that is not worn out may start a debate. However, we assumed that the software can also be continually exposed to potential bugs over time and that the software is also aging.

Second, random faults should be injected inside the software, and the definition for injected faults should be interpreted differently. The injected fault defined as a fault may not be recognized as a fault inside the software, and the failure weight may also be different because the injected fault has different effects on a software action.

Third, the failure rate to be used for the reliability evaluation formula should be defined. If any fault is injected in the location of the software and the fault detection coverage through the diagnostics software is calculated, the failure rate of the target software can be determined.

These issues are explained in detail as follows.

A. *Reliability calculation method*

A reliability is a way to express the probability that electronic components are continuously operated for a certain time. This is expressed as follows (3)(4):

$$R(t) = Pr(T \geq t) = 1 - Pr(T < t) = 1 - F(t) = 1 - \int_0^t f(t)dt \quad (3)$$

$$R(t) = 1 - F(t) = e^{-\lambda t} \quad (4)$$

F(t) is the failure cumulative distribution function and means the probability of malfunction within time t. It is expressed as follows (5)(6):

$$F(t) = Pr(T \leq t) = \int_0^t f(t)dt\,, t \geq 0 \quad (5)$$

$$F(t) = \int_0^t \lambda e^{-\lambda t}dt = 1 - e^{-\lambda t} \quad (6)$$

In addition, the *(t)* factor used in the failure cumulative distribution function of the system refers to the number of faults per unit of time. The most important factor is the failure rate *(t)* in the basic method for calculating the reliability. This is because the reliability calculation value is changed according to the number of faults in the system per unit of time. The failure rate calculation is as follows (7)(8)(9):

$$\lambda(t) = 1 - C \quad (7)$$

$$C = Pr(fault\ detected \mid fault\ existence) = {\sum FiDi}/{\sum Fi} \quad (8)$$

$$\lambda(t) = 1 - {\sum FiDi}/{\sum Fi} \quad (9)$$

There are various ways to calculate the failure rate expressed as a constant value. Among them, it is a general method that estimates the failure rate value using a probability analysis method using the test data and analysis data and calculates the reliability using the estimated values. The test data and the analysis data should be sufficient for the accuracy of the probability. However, there is a limitation in extracting the test data from the situation in which the controller is applied to the safety system and is operated. The samples also are very small, and thus it is inappropriate for use in statistics. In addition, determining the test result as a representative value of the failure rate is not rational because tests performed in the development process is not guaranteed. Random faults are injected in the software of the developed completed controller to escape the weakness, and it can then be possible to obtain the reliability of the software after calculating a failure rate using the diagnostic functions of the system.

B. *Definition of SW failure in Controller*

Because software within the controller in an NPP conducts the same program repeatedly, the area for the software has been limited. Thus, the definition for the fault within the controller is necessary. Because the fault occurs in the previous step of importing the system failure, even if a fault occurs, the system is not unconditionally experiencing a failure. By affecting the program or system task performing this safety function, the faults may or may not generate a system failure. For example, if even a specific area of the memory has been adhered to the value of bit 0, if the application using the memory uses the specific area as space for a constant, the integer value for the software does not change because the most upper bits remain as the value of bit 0. When the decimal value 15 is saved in the 10 bit space of

integer type memory, the binary value stored in that space will be "0000001111". At this time, the upper 4 bits will always be stored as a value of zero. Although the value of the upper 4 bits fixed to a value of zero by external shock, it does not affect the safety operation of the software. These faults in the controller in an NPP should not be treated as faults. It is necessary to distinguish whether the application in the controller uses the location or not when calculating the failure of the controller in the NPP. The fault in the position where the application program is not utilized is excluded from the faults. If this fault does not affect the safety program, it is realistically difficult to detect the fault and it is also not easy to develop a diagnostic program that can detect the faults in the unused portion. In the case of implementation with a complex diagnostic algorithm, the real-time detection of the fault is not guaranteed, and the detection function may not be properly conducted because much time and a high cost are needed for its operation. The effectiveness of the fault is given depending on whether the fault can affect the operation of the software in an NPP.

C. Fault effect factors

There are some factors to be considered in order to determine the fault using a fault detection function. The fault coverage may be computed differently since the location, the type, and the nature of the faults are different individually. The fault factors for the software in an NPP are as follows.

Fault ∈ {type, duration, location, weight, recovery}

The type, duration, location, weight, and recovery ability are the factors for the faults. In particular, the weighting factor may have the greatest impact on the calculation of the fault detection coverage of the controller in an NPP. The recovery ability is not important in the controller in an NPP since a diversity protection system will be operated when the fault is detected. We focus on the fault detection coverage capability.

- Fault Type = stuck-0 fault，stuck-1 fault

The fault type is stuck-0 or stuck-1. A software program is operated in hardware memory and the input and output of the data are also utilized in the memory space. An action for injecting a fault occurs in the memory and the memory bit can then be stuck-0 or stuck-1. A memory fault injected in the hardware has one of the two corresponding fault types, and thus, the fault type of the target bit is determined according to a probability of 1/2.

- Fault Duration

The duration degree of a fault is one of the attributes for defining the fault. An injection fault may be lasting as a permanent fault. Another fault may be recovered to a normal state over time, although it occurs intermittently. In this study, we only consider a permanent fault and not an intermittent fault.

- Fault Location

The location of the fault is one of the attributes for quantifying it. It is important to determine whether a random fault is injected in any position. A random injection fault affects the quantification of the failure depending on whether it is located on the most significant bit or the least significant bit. The location of the fault can be defined as the weighting factor.

- Fault Weighting

Operating system software running on the safety controller in an NPP repeats the same operation, performs a calculation using the data received from the communication in repeated operations, and performs diagnostic operations. The code and data area of the accessed memory are fixed during one cycle of the application program. However, the number of accesses are different from each other. It is reasonable to assign a weight in accordance with the number of accesses because a fault in the memory space where can access frequently increases the probability, which can affect the safety operation.

IV. EXPERIMENT

An experiment for calculating the failure rate of the software in consideration of the proposed method is progressing. Until now, the memory space that a software application can access was classified according to the access count. This is shown in Figure 1.

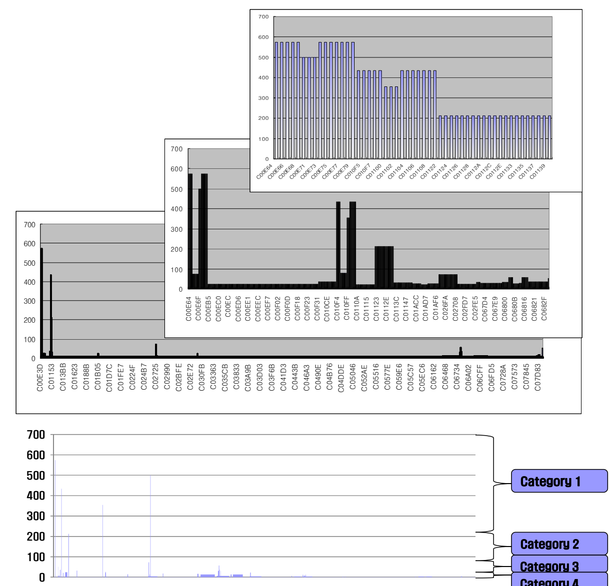

Figure 1. SW execution path and categorization.

To gain the first weighting factor related to the Fault Location, we addressed the random memory spaces that are utilized by a software program.

- Fault Injection Memory Address: 0x00C00E64 ~ 0x00C01139
- Fault Location: 0~31 bit
- Fault Type: stuck-0

Figure 2 shows the error effect statistics according to bit position in physical address.

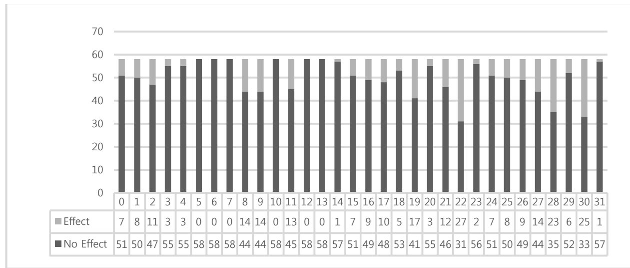

Figure 2. Error effect according to bit position of address.

Then, we can acquire the normalized weighting value using experimental sample data. Table 1 shows the normalized value depending on each bit position in address.

TABLE I. NORMALIZED VALUE OF BIT POSITION

Bit position	0	1	2	3	4	5	6	7
	8	9	10	11	12	13	14	15
	16	17	18	19	20	21	22	23
	24	25	26	27	28	29	30	31
Normalized value	-0.146	-0.013	0.389	-0.682	-0.682	-1.084	-1.084	-1.084
	0.791	0.791	-1.084	0.657	-1.084	-1.084	-0.950	-0.146
	0.121	0.255	-0.414	1.193	-0.682	0.523	2.532	-0.816
	-0.146	-0.013	0.121	0.791	1.996	-0.280	2.264	-0.950

We will estimate the software reliability using fault weighting value and the failure rate by the diagnostics software. The experimental results will be released in a future work, after the tests are completed.

V. CONCLUSION

We tried to calculate the software reliability of the controller in an NPP using a new method that differs from a traditional method. It calculates the fault detection coverage after injecting the faults into the software memory space rather than the activity through the life-cycle process. It is possible to calculate the software reliability when obtaining the failure rate and utilizing the existing calculation method. We attempt differentiation by creating a new definition of the fault, imitating the software fault using the hardware, and giving a consideration and weights for injection faults.

ACKNOWLEDGMENT

This paper was supported by the Ministry of Science, ICT (Information and Communication Technology) & Future Planning, Korea.

REFERENCES

[1] BTP-7-14, Guidance on software reviews for digital computer-based instrumentation and control system. NUREG-0800, Standard Review Plan: branch technical position 7-14, Revision 5, Nuclear Regulatory Commission.

[2] The Institute of Electrical and Electronics Engineers, Inc., “Standard Criteria for Digital Computers in Safety Systems of Nuclear Power Generating Stations,” IEEE 7-4.3.2.

[3] K. C. Kwon and M. S. Lee, “Technical Review on the Localized Digital Instrumentation and Control Systems,” Nuclear Engineering and Technology, vol. 41, no. 4, 2009, pp. 447-454.

[4] Gaurav Aggarwal and V. K Gupta, “Software Reliability Growth Model,” International Journal of Advanced Research in Computer Science and Software Engineering, vol. 4, 2014, pp. 475-479.

[5] H. S. Eom, G. Y. Park, H. G. Kang, and S. C. Jang, “Reliability assessment of a safety–critical software by using generalized Bayesian nets,” 6th International Topical Meeting on Nuclear Plant Instrumentation, Control and Human Machine Interface Technology, Knoxville, Tennessee 2009.

[6] H. G. Kang, “An Overview of Risk quantification Issues of Digitalized Nuclear Power Plants Using Static Fault Trees,” Nuclear Engineering and Technology, vol. 41, 2009, pp. 849-858.

[7] J. Duraes and H. Madeira, “Emulation of software faults, a field data study and a practical approach,” IEEE Trans. Softw. Eng,. vol. 32, no. 11, 2006, pp. 849-867.

[8] M. C. Hsueh, T. K. Tsai, and R. KIyer, “Fault Injection Techniques and Tools,” IEEE Computer, vol. 30, no.4, April 1997, pp. 75-82.

[9] Jean arlat et al., “Fault Injection for Dependability Validation: A Methodology and Some Applications,” IEEE Trans. On Soft. Eng., vol 16, no.2, Feb 1990, pp. 166-182.

[10] G. Choi and R. Iyer, “Focus, An Experimental Environment for Fault Sensitivity Analysis,” IEEE Trans. On Computers, vol.41, no.12, December 1992, pp. 1515-1526.

[11] Y. Yu, “A perspective on the state of Research on Fault injection techniques,” Research Report, University of Virginia, May 2001.

[12] PATENT, “Fault mode apparatus and method using software,” 10-1222349, The Korean Intellectual Property Office, 2013.

[13] H. Madeira, D. Costa, and M. Vieira, “On the emulation of software faults by software faults by software fault injection,” Proceedings of International Conference on Dependable Systems and Networks, 2000, pp. 417-426.

[14] S. Richter and J. Wittig, “Verification and Validation Process for Safety I&C Systems,” Nuclear Plant Journal, May-June, 2003, pp. 36-40.

[15] B.A. Gran and A. Helminen, “The BBN methodology: progress report and future work. OECD Halden Reactor Project,” HWR-693, 2002.