The Design and Implementation of Testable Integration Architecture for Complex Distributed Systems

B. Makoond[a,b]; S. Ross-Talbot[b]; S. Khaddaj[a]
(a) Faculty of Computing, Information Systems and Mathematics, Kingston University, Kingston Upon Thames, KT1 2EE, UK
(b) Cognizant Technology Solutions, Haymarket House, 28-29 Haymarket, London SW1Y 4SP
Bippin.Makoond@cognizant.com; Steve.Ross-Talbot@cognizant.com; S.Khaddaj@kingston.ac.uk

Abstract

Testable Integration Architecture (TiA) provides a framework and discipline that blends the techniques of deductive modelling with inductive modelling. The rationale for the TiA is to address the limitations of deductive modelling, i.e. the classical software engineering tools such as UML, ERD and DFD to describe the dynamic aspects of communication models. Since those artefacts are static by nature, they fail to model the dynamic behaviour of participants communicating across complex distributed systems. TiA exploits the capabilities of formal methods such as pi-calculus (inductive modelling) to formally describe the act of communication that can be mathematically expressed, hence exercised on a simulation engine to reveal the dynamic dimension of the design. In so doing, software engineers can automatically walk through a series of models to identify defects and ambiguity at the early stage of the life cycle. The motivation of implementing TiA is to reduce the cost of quality in software development life cycle by ensuring early defect detections and reduced defect injection.

Keyword: Testable Architecture, Distributed Systems, Quality Attributes.

1. Introduction

It is very often argued that Software Engineering within distributed system is an engineering of complex system. According to Gödel incompleteness theorem, a complex system can be defined as one that can only be modelled by an infinite number of modelling tools (Chai71). The development of distributed systems in domains like telecommunications, industrial control, supply-chain and business process management represents one of the most complex construction tasks undertaken by software engineers (Jenn01) and the complexity is not accidental but it is an innate property of large systems (Sim96).

In distributed systems we observe emergent behaviour since logical operations may require communicating and multi channel interactions with numerous nodes and sending hundreds of messages in parallel. Distributed behaviour is also more varied, because the placement and order of events can differ from one operation to the next. Modelling the interactions of distributed system is not straight forward and inherently demands a multi-disciplinary approach and a change in traditional mindset to be resolved.

This paper starts with a brief discussion of software modelling concepts and techniques. Then, the main idea behind the proposed testable integration architecture is presented. This is followed by a case study for large communication model of business critical systems together with some experimentations and observations. Finally, a summary of the finding is presented.

2. Modelling Concepts and Techniques

Unlike many engineering fields, software engineering is a particular discipline where the work is mostly done on models and rarely on real tangible objects (Oud02). According to Shaw, (Shaw90), Software engineering is not yet a true engineering discipline but it has the potential to become one. However, the fact that software engineers' work mainly with models and a certain limited perception of reality, Shaw believes that the success in software engineering lies in the solid interaction between science and engineering. In 1976, Barry Boehm (Boeh76) proposed the definition of the term

Software Engineering as the practical application of scientific knowledge in the design and construction of computer programs and the associated documentation required to develop, operate, and maintain them. This definition is consistent with traditional definitions of engineering, although Boehm noted the shortage of scientific knowledge to apply.

On one hand, science brings the discipline and practice of experiments, i.e. the ability to observe a phenomenon in the real world, build a model of the phenomenon, exercise (simulate or prototype) the model and induce facts about the phenomenon by checking if the model behaves in a similar way to the phenomenon. In this situation, the specifications of the phenomenon might not be known upfront but induced after the knowledge about the phenomenon is gathered from the model. These specifications or requirement are known a posteriori.

On the other hand, engineering is steered towards observing a phenomenon in reality, deducing facts about the phenomenon, build concrete blocks; structures (moulds) or clones based on the deduced facts and reuse these moulds to build a system that mimics the phenomenon in reality. In this situation, the specifications of the phenomenon are known upfront, i.e. deduced before even constructing any models, whilst observing the phenomenon. The process of specifying facts about the phenomenon is rarely a learning process, and requirements are known a priori.

The scientific approach is based on inductive modelling and the engineering approach is based on deductive modelling. Usually in software engineering we are very familiar with the deductive modelling approach, exploiting modelling paradigm such as UML, ERD, and DFD that are well established in the field. However, the uses of inductive modelling techniques are less familiar in business critical software engineering, but applied extensively in safety critical software engineering and academia. Typically, inductive modelling techniques are experiments carried out on prototypes, or simulation of dynamic models which are based on mathematical (formal methods), statistical and probabilistic models. The quality of the final product lies in the modelling power and the techniques used to express the problem. As mentioned earlier, we believe that the power of the modelling lies in the blending of the inductive and deductive modelling techniques.

The rationale of integrating inductive modelling techniques within the domain of our study is due to the elements of non-determinism, emergent behaviours, communicational dynamics which are those parts of the problem that cannot be known or abstracted upfront i.e. a priori. These elements differs from those parts of the problem that can be abstracted from a priori based on experience and domain knowledge, which are normally deduced and translated into structures or models (moulds) i.e. using deductive modelling techniques.

Inductive modelling techniques require a different approach of addressing the problem attributes. In these circumstances, we tend to believe that the requirements are false upfront, and the objective is to validate these requirements against predefined quality attributes. To do so, we build formal models (formal methods) to mimic the functionalities of the suggested requirements and run the models (dynamically) to check if the models conform to the expected output and agreed quality. The modelling tools are dynamic in nature, and very often they offer themselves very easily to simulation engines and formal tests that allow system designers to run and exercise the designs, to perform model validation and verification. Through several simulation runs, the models are modified, adjusted and reinforce until they match, to certain level of confidence, the quality attributes.

Moreover, modelling distributed software architectures require a multidisciplinary approach to modelling i.e. that there are several ways of modelling the problems attributes and we were required to combine several of these approaches and models as shown in Figure 1.

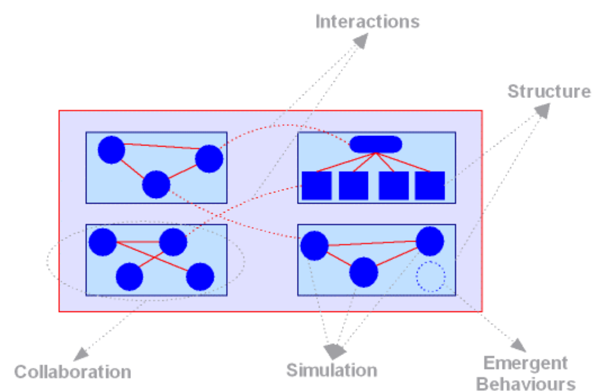


Figure 1 Multi-disciplinarism in Modelling

Arguably, there are two types of modelling approaches, inductive and deductive, within the field of software engineering.

- Deductive Modelling includes the aspect of structural, functional and collaborative designs and is commonly used in classical software engineering, such as Class Diagrams, Sequence Diagrams, Object Diagrams, Entity Relationship Diagrams (ERD), Data Flow Diagrams (DFD), Flow Charts, Use Cases, etc…
- Inductive Modelling, are critical dynamic modelling techniques that primarily characterise the aspect of non-determinism within a system mainly arising from the occurrence of emergent behaviour and interactions. Commonly used techniques are formal methods (e.g. Testable Integration Architecture), simulations and probabilistic models of the software artefact.

3. Testable Integration Architecture (TiA): Blended Modelling Approach

For many years, computer scientists have tried to unify both types modelling techniques in order to capture the several facets of the distributed communication systems and demonstrate the power of modelling to develop software artefacts of high quality.

The development of distributed systems and applications is a complex activity with a large number of quality factors involved in defining success. Despite the fact that inductive modelling is scientifically thorough for analysing and building quality engineered systems, it brings additional cost into the development life cycle. Hence, a development process should be able to blend inductive and deductive modelling techniques, to adjust the equilibrium between cost (time and resources) and quality. As a result, the field of software process simulation has received substantial attention over the last twenty years. The aims have been to better understand the software development process and requirements and to mitigate the problems that continue to occur in the software industry which require a process modelling framework.

3.1 Requirement Engineering

Our proposition of implementing a blended modelling approach in software engineering starts with the process of the requirements analysis. Requirements are usually categorised into two general types; there are the functional requirement and the non-functional requirements. The IEEE defines functional requirement as a requirement that specifies a function that a system/software system or system/software component must be capable of performing. These are software requirements that define behaviour of the system, that is, the fundamental process or transformation that software and hardware components of the system perform on inputs to produce outputs. Non-functional requirement, on the other hand, in software engineering is a requirement that describes not what the software will do, but how well the software will perform its task, e.g. the SLAs of the system, software performance; software external interface requirements, software design constraints, and software quality attributes.

In this study, we extended on these two general styles to propose a new requirement framework that is composed of four different requirement styles. We have broken down the functional requirement styles into 1) the data and structural style, 2) functional and behavioural style and 3) the communication style. The non-functional requirements are referred to as the quality attribute style in this study.

Since, the class of problems has many dimensions and we need to identify each of these dimensions so that the right modelling tools and techniques are applied for the right problem attribute. We created a classification technique that categorises the problem attributes in different requirements styles; hence the selection of modelling techniques to represent the requirements can be based on the characteristics or styles of the requirements. Depending on the styles, we define the modelling specifications and employ the appropriate tools. The four different styles are explained as follows:

Data and Structural Style

This style holds the part of the system that describes the data, i.e. the properties of the data, the types of data, the data structure and its organisation or order within the system. The task of the data styles is to primarily separate the data from the process of treating of data. The classical modelling tools available to handle requirements within this style are entity relationships diagram, class diagram, object diagram and component based diagram.

Functional and Behavioural Style

This style defines the functions of the system without any regards to the class these function belong to. The main objective of the functional styles is to describe the input, the process and describe the expected output of a process. The behavioural part requires

tools such as state charts and Petri Nets that illustrate the change in state of a program when a particular transition of that program has occurred. Classical tools of the functional and behavioural styles are pseudo code, B machines, Z notations, collaboration diagrams, flowcharts etc.

Communication Style
The communication style handles the complex features of system interactions than entail both the inter-communication amongst external systems and the intra-communication of internal components. The style provides the specifications of modelling the aspect of communication to capture how the different parts of the systems communicate using different styles of communication and the type of communication medium in use. These modelling tools are usually dynamic that naturally offers themselves to simulation. The dynamism is essential to describe emergent behaviours within the communication model. The tools to model the attributes of the communication style are simulation tools such as CDL (pi-calculus), Estelle, SPIN, and prototyping.

Quality Attributes Style
Quality Attributes Style incorporates methods of modelling the non-functional requirements, i.e. the SLAs. According to Somerville, (Somm01), different kinds of non-functional requirements exist; ranging from product requirements, organisational requirements to external requirements. The product requirements specify the behaviour of the product, e.g. execution speed, robustness and reliability. Organisational requirements are originated from the organisation's policies and procedures, e.g. process standard used and implementation requirements. All requirements that come from external factors to the system and its development process belong to the external requirements. One type of external requirement is security. However as explained in the discussions on software requirement engineering, (Somm01, Lau02, Krem98), the problem with non-functional requirements is the difficult to verify and very often they interact and conflict with functional requirements and with each other.

3.2 Blended Modelling and Testable Integration Architecture

The method of Testable Integration Architecture (TiA) addresses the problem of quality modelling enabling software analysts to translate functional requirements and SLAs into dynamic modelling artefacts that can be used to measure ambiguity and conformance to requirements, hence validated the design.

Each requirement styles correspond to a particular view of the system and address a particular aspect of the Class of Problems. However, we observed during the study, the different views are not independent but interdependent. The multiplicity or variety of styles helps to enhance understanding the class of problem so as to reduce ambiguity.

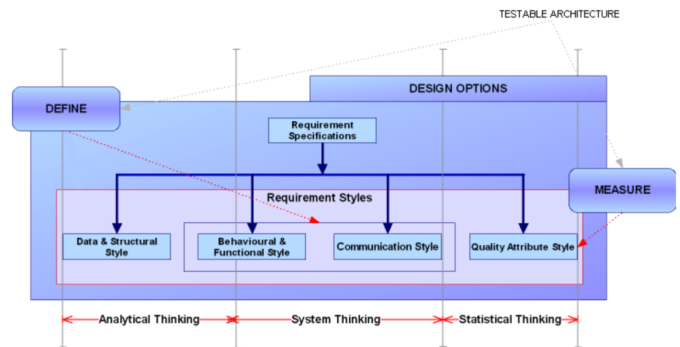


Figure 2 Testable Integration Architecture supporting the Blended Modelling Approach

The diagram depicted in Figure 2, is a high level model of the application of TiA to 1) classify requirements, 2) design and simulate the dynamic model of the requirements, and 3) validating the requirements and design. Firstly, TiA acts as an analytical tool to decouple the requirement into their own style for analysis; secondly as a systemic modelling tool, providing a global view of the communication model and dynamic behaviour, founded on a robust simulation engine and thirdly, it provides type checks and statistical results that can be consumed by statistical methods to test the design for quality i.e. checking the design against the non functional requirements.

In

, we provide an end to end process life cycle of the blended modelling approach focussing on the capability of Testable Integration Architecture. The diagram shows the exploration of a multitude of tools which are required to comprehend the requirements of a complex system.

At the elicit phase, requirements are gathered and invented and at this particular point in time, those requirements are untreated. At the decompose stage, we employed techniques of the House of Quality to

break down the requirement into their corresponding styles which leads us to the classify stage.

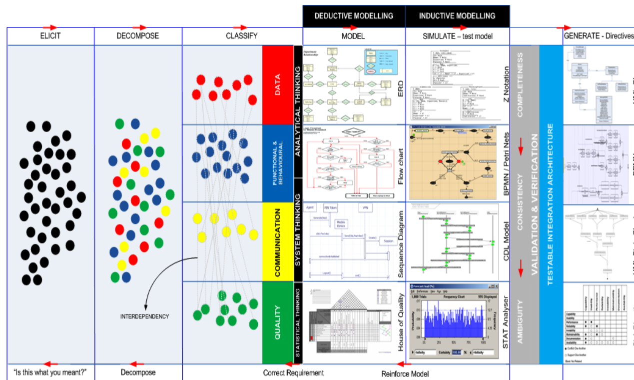


Figure 3 The Blended Modelling Approach

In a Blended Modelling approach, there are various styles to represent a system which is characterised by different requirement styles; 1) the data style are those requirements that answer to the question of what; 2) the functional and behavioural style which represents the logical function or the business process are those requirements that answer to the questions of how and when; 3) the communication styles are those requirements that answer to the question of where and 4) the quality styles which is a cross functional styles and answer to the question of how well each of the 3 aforementioned styles is expected to perform. As

depicts, theses styles are not independent but interdependent. Each style is transformed into models using modelling tools that were design to specifically describe the characteristics of individual requirement styles. We adopt both the deductive and inductive tools in our blended approach since we want to test and validate the models or requirements as early as possible so as to reduce the number of defects leaking to the implementation phase of the life cycle. The inductive modelling tools add scientific rigour and transform the static structural models into a formal model using formal methods and hence test can be compiled on those design; e.g. transforming a flow chart (describing a logical function) into Coloured Petri Nets or sequence Diagram into CDL, so that these models are tested and simulated for verification and validation. The table in

Requirement Styles	Verification Model	Structural Artefacts (Deductive)	Controllable Artefacts (Inductive)
Data and Structural	Completeness of Structure	Entity Relationship Diagram	<Z Notation> [TiA]
Functional and Logical	Consistency	Flow charts and scenarios	<BPMN 2.0 Petri Nets> [TiA]
Communication and Behavioural	Consistency	Sequence Diagram and Collaboration Diagram [UML]	<Pi Calculus, CDL, Scribble> [TiA]
Quality	Ambiguity	Requirement Catalogue Entry Quantitative Specifications	<House of Quality> [TiA]

Figure 4 shows the translation from deductive modelling tools (structural and static) to inductive modelling tools (formal and dynamic).

Requirement Styles	Verification Model	Structural Artefacts (Deductive)	Controllable Artefacts (Inductive)
Data and Structural	Completeness of Structure	Entity Relationship Diagram	<Z Notation> [TiA]
Functional and Logical	Consistency	Flow charts and scenarios	<BPMN 2.0 Petri Nets> [TiA]
Communication and Behavioural	Consistency	Sequence Diagram and Collaboration Diagram [UML]	<Pi Calculus, CDL, Scribble> [TiA]
Quality	Ambiguity	Requirement Catalogue Entry Quantitative Specifications	<House of Quality> [TiA]

Figure 4 Transforming deductive to inductive

Finally, the last cycle of the blended modelling approach, TiA generates the design directives of the validated artefacts which are submitted to the developers for implementation.

In this paper we focus on the communication styles of a given system. When it comes to modelling the interaction and communication of Distributed System, Choreography Description Language (CDL) (Yang06) is one of the most efficient and robust tool. CDL forms part of Testable Integration Architecture, hereafter TiA, and is based on pi calculus (Miln99), which is a formal language to define the act of communicating.

3.3 The Language of Pi Calculus

Although formal methods exist such as Petri Nets (Pet62), B Methods, Z Notations and lambda calculus that are used to unambiguously describe software requirements. However when it comes to describing distributed concurrent interactions of several participants, they encounter major difficulties since they were not design to do so. Lambda calculus was designed for parametric description of passing arguments across functions; Z Notation was designed to classify and group attributes of the problem domain into logical sets; and B Methods was

designed to describe requirement into logical sets and consistent machines.

Pi-calculus is one element in a set called Process Calculi. The distinguishing feature between pi-calculus and earlier process calculi, in particular Calculus of Communicating Systems (CCS)(Miln80a) and the work done on Communicating Sequential Processes (CSP) (Hoare85), is the ability to pass channels as data along other channels.

Pi calculus is a formal language that uses to concept of channels and naming to describe interactions and fits comfortably in the problem domain of distributed systems. The pi-calculus is a model of concurrent computation based upon the notion of naming (Miln93). The syntax of pi-calculus enables one to model processes, parallel composition of processes, and communication between processes through the concept of channels. A channel is an abstraction of a communication link between two participants, the same way a process is the abstraction of a given thread of control. In the pi-calculus language there are a given set of constructs where all possible concurrent behaviour can be written. The following paragraph explains these constructs.

Let P and Q denote processes, then P |Q denotes a process composed of P and Q running in parallel. The construct a(x).P models a process that waits to read a value x from the channel a, and then, having received it, behaves like P. The construct Ā<x>.P denotes a process that first waits to send the value x along the channel a, and then, after x has been accepted by some input process, behaves like P. The construct (va)P ensures that a is a fresh channel in P. The construct !P denotes an infinite number of copies of P, all running in parallel. The construct, P + Q denotes a process that behaves like either P or Q. 0 denotes the inert process that does nothing.

TiA abstracts the given set of constructs in pi-calculus and provides a language through a series of methods and logical sequences that are presented in a unified toolset. The latter facilitates the modelling and simulation of communication in concurrent systems. In order to achieve rigour in the power of modelling, TiA exploits the capability of CDL as a framework to model global message flows and the subsequent impact of communication on local behaviours, which is defined by pi-calculus. The work done by Carbone et al (Carb06), show formal description in the global calculus, has a precise representation in the local calculus. As a result, unlike other modelling frameworks, TiA is not limited to deductive and static modelling techniques, as it uses pi calculus based on non-deterministic models, that are well known within the academic world, but not yet of a common use within industry. In fact TiA acts as a natural "glue" to blend the various modelling approaches providing a framework with the primary objective of removing the characteristic of ad-hocness and ambiguity within the modelling Process.

Using TiA, the formal description of the requirements can be translated into different types of modelling tools starting with dynamic modelling tools (inductive modelling) such as Coloured Petri Nets (CPN) and prototyping, then moving to event based modelling tools such as State Chart Diagrams and Sequence diagrams and finishing with structural modelling tools (deductive modelling) such as class diagrams. Throughout the translation process, the specifications and requirements can be tested, validated and reinforce.

4. Case Study: Testable Integration Architecture used in Large Communication Model of Business Critical Systems

In the case study, we focus on the fundamental problem of underwriting within a global insurance group, which includes the characteristics of Underwriting Workflow System, Policy Manager, Document Management System and the Integration Layer. The aim is to demonstrate how TiA is used to reinforce the power of modelling by avoiding classical modelling pitfalls, defining traceability across the lifecycle, providing a reference model through iterations, and addressing defects at early stage, hence increasing the maturity of the process model.

As we mentioned earlier, the design approach employs both the deductive and inductive modelling techniques, and TiA employs a formal method, Pi-calculus that provides the ability to test a given architecture, which is an unambiguous formal description of a set of components and their ordered interactions coupled with constraints on their implementation and behaviour. Such a description may be reasoned over to ensure consistency and correctness against requirements.

4.1 The Requirement of the Communication Model

Prior to designing the communication artefacts in TiA, we observed the requirements that have been gathered or invented during the requirement analysis phase of the Software Development Cycle (SDLC). Due to the size of the requirement catalogue, in the context of this paper, we focus on those parts of the requirements that demonstrate the character of communicative behaviour or the act of communicating.

The architecture provides communication management and enablement of external systems deployed over an ESB layer, conforming to the principle and discipline of SOA. The architecture diagram, depicted in Figure 6, is a representation of the requirements of the communication between an Underwriting Workflow System and a Policy Manager (denoted as GWS in the illustrations). The communication is handled by the integration layer, employing BizTalk as technology and the Underwriting Workflow System is implemented using Pega PRPC.

There are two primary motivations behind the use of TiA in the context of this work:

1. Firstly, we employ TiA to develop several dynamic representations of the communication model in between the 1) Underwriting Workflow System; 2) Policy Manager and 3) BizTalk supporting the Integration Layer, in order that when these models are simulated, the results produced can be tested and verified against the requirements.

2. Secondly we employed TiA in the given problem domain, is to achieve a model of communication that can evolve, consequently allowing BizTalk to move from being purely an EAI to the capability of an ESB wherein heterogeneous types of communication which includes external participants will be possible. Such conversation will be with Document Management Systems, Claims Repository Service, external Rating Services and others.

In our problem domain, BizTalk maps the message of Pega PRPC, hereafter Pega, to the legacy Policy Manager. This is carried by transforming the data structure of the Pega messages into the data structure of native Policy Manager. There are 3 generic types of communication that describes the conversation between Pega and BizTalk.

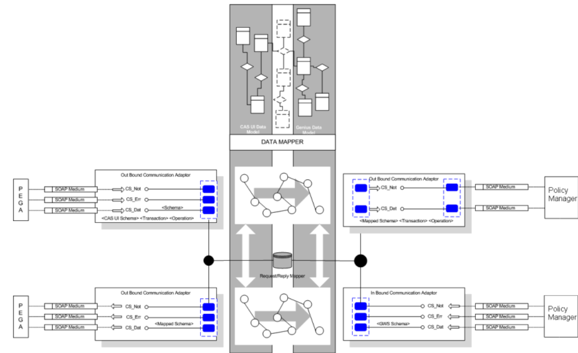
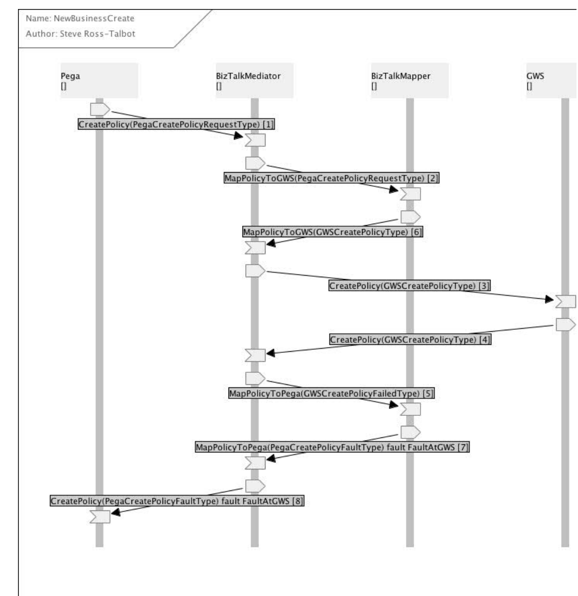


Figure 6 Communication Model

The



communication model illustrates 3 communication types 1) notification, error and data, expressed as <CS_Not>, <CS_Err> and <CS_Dat> respectively which are transported from Pega to BizTalk. BizTalk accesses the data mapping schema and transform the incoming schema into response schema which is agreed by the Policy Manager. The Data Mapper is logically represented by the ERD

From BizTalk to the Policy Manager, there are two types of communication which are 1) notification, <CS_Not> and 2) Data, <CS_Dat>. The communication model represented follows an asynchronous mode, which is handled by the Request/Reply map repository. The latter holds the state that assigns the corresponding response from the Policy Manager to a Request from Pega. There is a polling mechanism to notify Pega that a response has been received for a corresponding request.

There are 3 return communication types from the Policy Manager to BizTalk which are <CS_Not>, <CS_Err> and <CS_Dat>. The latter holds the data which is required by Pega to update any underwriting transactions. As we modelled the communication using TiA, it has been observed that the existing legacy Policy Manager interface does not differentiate between success and failure response, hence there is no separation of identity between the error and success, which complicates the design of the integration layer. The design flaw has been identified whilst validating and type checking the communication model with TiA. This has lead to some mistake proof mechanism within BizTalk to manage error and trace the error back to the presentation layer, i.e. General Underwriting System. BizTalk has to transform the Policy Manager schema into a structure agreeable by Pega. The communication medium employed across Pega, BizTalk and Policy Manager is SOAP.

4.2 Implementing Testable Integration Architecture (TiA)

The process starts at the requirement gathering phase, where TiA is used to identify the core aspects of the communication which are in our context, the Pega component, The BizTalk component and Policy Manager (PM), as shown in Figure 7.

Figure 7 Requirement communication model

At the very early stage of design, while validating the communication with TiA through formal checking, it has been observed that the BizTalk component includes two primary modules, which is required to be modelled separately, and these are the Mapper component and the Mediator component respectively. This is a typical problem of separation of concerns. The separation showed that the mediator service is solely concerned with the orchestration of the communication model whereas the mapper service is related to the data modelling which ought to be abstracted to the problematic of Canonical Data Model within an ESB.

The separation of concern to abstract distinct services (mediator and mapper services) has been possible because the TiA tool suite requires one to model requirements as conversations between concrete behaviours. It forces a separation into those behaviours by making it inconvenient to model otherwise, i.e. one participant communicating to itself. TiA has the innate property to separate behaviours out.

Consider a process P which has a conversation with P. Then P is split into P' and P'' and the conversation is modelled as P' and P''. As P communicates with P, the behaviour of P as a participant, changes. As a result P has two personae, characterised as P' and P'' respectively. Hence TiA engages the designers into a style of modelling that removes message likes within behaviours and adds an additional behaviour (which may or may not be the same participant). This reflects good design by forcing the requirements, and then the model, into clearly delineated behaviours, which is a fundamental practice within the problematic of Service Identification for Service Oriented Architecture.

Using classical modelling techniques, purely static design such as sequence diagram, this dichotomy is not enforced and would have been missed in the requirement phase and only be found at the late stage of design or coding. It is also possible that the separation would have been missed completely, adding overheads and reworks to preserve the characteristic of extensibility to the architecture.

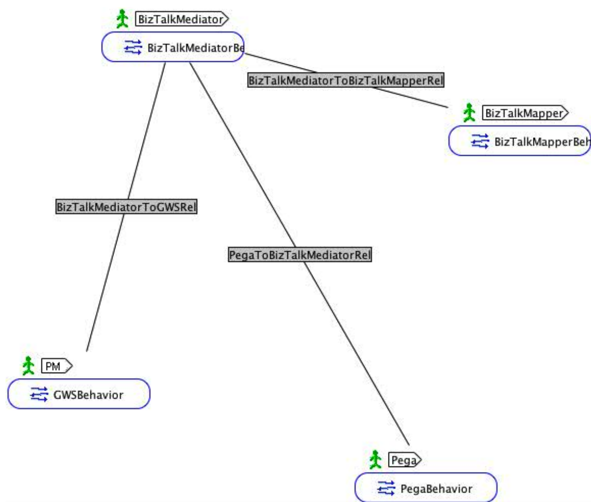


Figure 8 Conversation Model

Whilst requirements are gathered, a model of the conversation within problem emerges as shown in

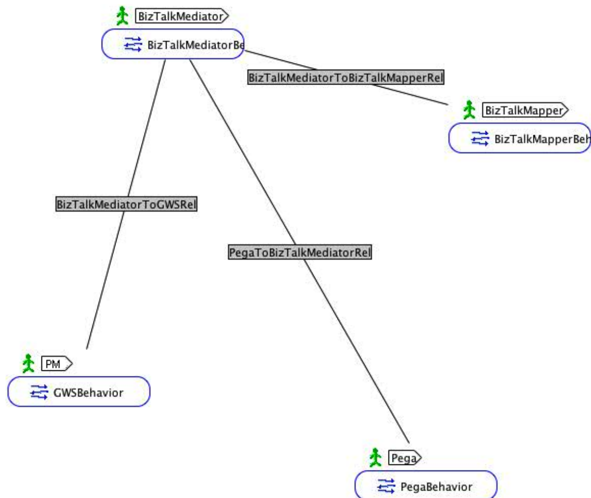


Figure 8. This is static diagram that simply lays out the roles, the swim lanes (see Figure 7), and who can talk to who. This enables us to manage the conversation in the system and to also extend the model to add new components and test if the communication model still holds when new participants are added.

The next step is to bind the model in Figure 7 to a choreography, which will enable us to type check the model against the requirement in order to validate the model and remove ambiguity in the requirements for the communication model. The choreography is shown in Figure 9.

Figure 9 Architecting the Design (Partial view of the Dynamic Choreography model)

The binding process involves the process of referencing the model in the requirement and binding the interactions. The binding process also has the effect of filling in some of the missing information on identity and business transactions.

With a bound model, the choreography in Figure 9 can be exercised in order to prove the model against the architectural parameters which are derived from the given set of requirement as shown in Figure 10. The model shows the participants which are Pega, conversing with the BizTalk's mediator service, the mapper service (for data transformation) wherein data is finally be passed to the Policy Manager participant.

4.3 Simulation & Observation

During the test of the architecture, the proof goes green (see Figure 10) if the configuration and parameters or more precisely the types of the interactions are correct and should it be red, the proof reveals that the model deviates from the requirements, highlighting the defects. The binding and rendering of the bound requirements provide very precise documentation for implementers.

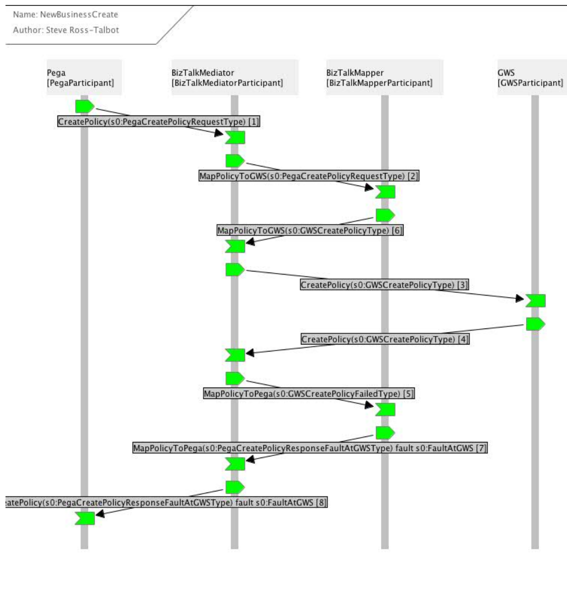


Figure 10 Proving the Communication Model

Hence for each interaction, we can clearly observe the meaning of identity, the meaning of the type for each identity (the token or tokens) and the Xpath expressions which is executed during the simulation over the example message, in our case the request xml of Pega and the Policy Manager Process UW xml, return the appropriate values. This is shown in Figure 12.

Ref	Identity Tokens	Identity Values	Query Expressions	Message File
[1]	PolicyIdentity	UKCASA00001108	//s0:Reference/text()	PegaCreatePolicyRequestType.xml
[2]	PolicyIdentity	UKCASA00001108	//s0:Reference/text()	PegaCreatePolicyRequestType.xml
[3]	PolicyIdentity	UKCASA00001108	//s0:Reference/text()	GWSCreatePolicyResponseFaultType.xml
[4]	PolicyIdentity	UKCASA00001108	//s0:Reference/text()	GWSCreatePolicyResponseFaultType.xml
[5]	PolicyIdentity	UKCASA00001108	//s0:Reference/text()	PegaCreatePolicyResponseFaultGWSMappingType
[6]	PolicyIdentity	UKCASA00001108	//s0:Reference/text()	PegaCreatePolicyResponseFaultGWSMappingType

Figure 12 Identity of Interactions

The values returned during the course of a simulation, compiled within the xml output files as shown in Figure 12, accentuate the meaning of identity and types which provides a set of defined results that can be revised against the requirements gathered. This enables the decision makers to verify and validate the yield of the communication model against conformance to the requirement of the customers. Typically this exercise is run as a proof of concept. As a result the Business Analyst is empowered to walk through the simulation results with the clients, asking the vital question of: "is this what you meant?" The walk through process is inherently more robust than traditional design inspection exercise and formal reviews (). The reason is because TiA adds scientific rigour and is supported by the simulation engine and type checking protocol that can be run and re-run during the SDLC prior to any coding.

After the proof of the model is demonstrated, we believe that the model conforms to the pre defined requirements and many of the ambiguities in the requirements have been detected and consequently resolved at the requirement and design phase of the Software Development Life Cycle (SDLC). Then, in exploiting the capabilities of model generation, TiA provides us with a rich a proven set of artefacts such as UML designs and state-charts diagram of the model. In Figure 13, we show the state-charts generated from the proven dynamic models. This is typically the translation of the inductive models (the CDL model) to the more common deductive models (UML and BPMN). Then the course of the SDLC resumes with the normal route of the classical software engineering processes.

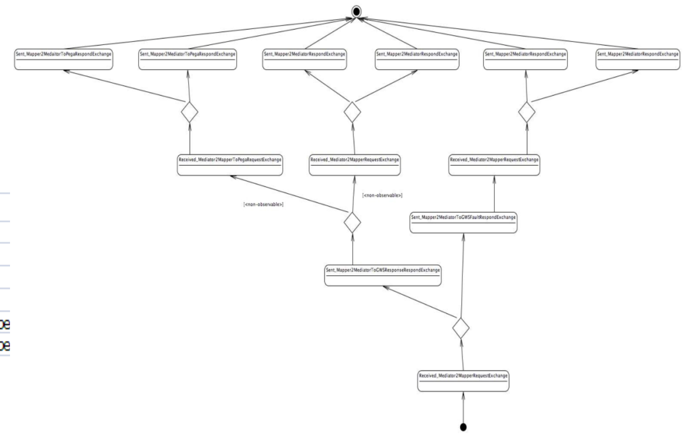


Figure 13 Generated UML Artefacts State Chart of the Underwriting System

During the Design exercise we exploited the feature of Business Process Execution Language (BPEL) generation from the TiA framework to generate artefacts that can be imported into a BPEL compliant orchestration tools. TiA guarantees that the BPEL preserve the state behaviour that is shown in the state charts (see Figure 13) for the given service.

In Figure 14, we present the generated BPMN artefacts from the TiA toolset. In this paper we focus

on the BizTalk Mediator Service of the Systems as most of the complex interactions are being coordinated in this layer. The BPMN model is derived from the refined and proven CDL Model.

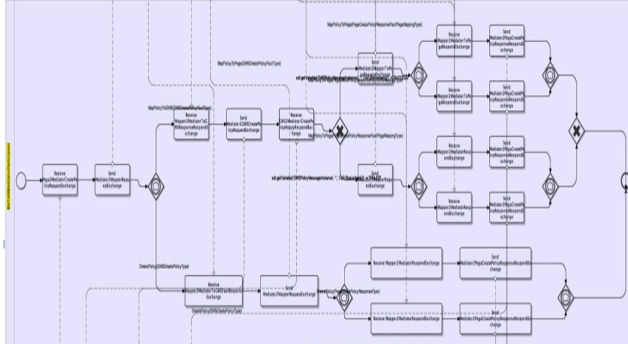


Figure 14 Generated BPMN Model - Focussing on the Processes of the BizTalk Mediator Service

The generated models along with auto-generated documentations are compiled into the design directives and coding principles that can be handed over to the software designer and the developers. The communication to these parties is founded on formal and mathematical checks which makes the design and the development of the system far less error prone.

5. Conclusion

In employing TiA, we were able to identify business and core service easily and test them against requirements for the mediator business service and mapper core service. We worked very closely with key decision makers to ensure a full understanding and gain agreement on requirements through inductive modelling of requirements and the collaboration model that is embodied in TiA. This allowed rapid turn-around with Business Analyst and reduced the overall design time.

Secondly we were able to detect errors both as conflicting requirements (reported back and then remediated with the stakeholders) and technical design errors prior to coding, the latter being the legacy Policy Manager's error handling problem. We were also able to simplify the design segmenting it and ensuring that it truly represented the requirements through TiA.

Finally, TiA enabled the generation of implementation artefacts, such as UML designs and state charts that were guaranteed to meet requirements and were of an order of magnitude more precise which reduced the communication need to ensure a high quality delivery. This is typically the capability of TiA to blend the inductive with the deductive modelling techniques.

The benefits of blending inductive modelling techniques with the deductive techniques yield to earlier defect detection resulted to reducing the transition time to architect the solution by generating precise technical contracts for implementation which is type checked, hence proven to be correct. In our case, we are able to achieve a 40 % efficiency to move from requirement to technical specifications. Moreover, the simulation exercise empowered us to check if the model is implementable against the given technology stack (BizTalk and Pega).

6. Reference

(Boeh76) Boehm B W, "Software Engineering", IEEE Trans. Computers, pp. 1,226 - 1,241, December 1976

(Carb06) Carbone M, Honda K, Yoshida N, Milner R, Brown G, Ross-Talbot S , "*A Theoretical Basis of Communication-Centred concurrent Programming*", PhD thesis, Imperial College, London UK, 2006

(Chai71) Chaitin G J, "*Computational Complexity and Godel's Incompleteness Theorem*", ACM SIGACT News, No. 9, IBM World Trade, Buenos Aires, pp. 11- 12, April 1971

(Hoar85) C.A.R. Hoare, "*Communicating Sequential Processes*", Prentice-Hall, 1985

(Jenn01) Jennings N R, "*An Agent-based approach for building complex software systems*", Communications of the ACM, Vol 44, No. 4, April 2001

(Miln80a) Milner R, "*A Calculus of Communicating Systems*", Lecture Notes in Computer Science, volume 92, Springer-Verlag, 1980

(Miln80b) Milner R, "*A Calculus of Communicating Systems*", Lecture Notes in Computer

Science, volume 92, Springer-Verlag, 1980

(Miln93) Milner R, “*The Polyadic pi-Calculus: A Tutorial*”, L. Hamer, W. Brauer and H. Schwichtenberg, editors, Logic and Algebra of Specification, Springer-Verlag, 1993

(Miln99) Milner R, “*Communicating and Mobile Systems*”, Cambridge Press, June 1999

(Oud02) Oudrhiri R, “*Une approche de l’évolution des systèmes,- application aux systèmes d’information*”, ed.Vuibert, 2002

(Pet62) Petri C A, "*Kommunikation mit Automaten*", PhD thesis, Institut f¨ur instrumentelle Mathematik, Bonn, 1962

(Shaw90) Shaw M, “*Prospects for an Engineering Discipline of Software*”, IEEE Journal, Carnegie Mellon University, 1990

(Sim96) Simon H A, “*The Sciences of the Artificial*”, MIT Press, 1996

(Yang06) Yang H et al, “*Type Checking Choreography Description Language*”, Lecture Notes in Computer Science Springer-Berlin / Heidelberg, Peking University, 2006